Rapid and efficient mixing in low Reynolds number microfluidic systems via short-distance flow through an electrospun nanofiber mat: A computational analysis

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering

Andrei Georgescu

August 3, 2012

Acknowledgments

First and foremost I offer my sincerest thanks to my advisor, Dr. Antje J. Baeumner, who for the past several years has been a exceptional source of expertise, advice, support, and motivation, and without whom this thesis would not have been possible. I not only appreciate the guidance she has provided in the capacity of an advisor, but also as a dedicated professor in whose classes I have been enjoyably enrolled both as a student and teaching assistant.

I wish to thank all members, past and present, of the Bioanalytical Microsystems & Biosensors laboratory for their help, suggestions, and motivatingly strong work ethic throughout the past several years.

I would like to acknowledge all of the fantastic and dedicated professors and staff in both the BEE and BME departments. I would especially like to thank Dr. James Bartsch, for his support during my time as an undergraduate; Dr. Dan Luo, for teaching an exceptional class that has been one of the most useful I have taken at Cornell; Dr. Ashim Datta, for his invaluable advice and suggestions on this project as well as for his dedication to his Heat & Mass Transfer class; and to all of the remaining professors in whose classes I have had the priviledge of being enrolled.

Lastly, and above all, I would like to thank and dedicate this thesis to my parents, Mihail and Daniela Georgescu, for their love, advice, and unconditional support.

Abstract

This report presents the development of a comprehensive method by which electrospun nanofibers and their mixing effects may be modeled with a combination of custom scripts and ANSYS 14.0 software. A customizable and entirely automated workflow is realized, through which accurate models and results may be generated without requiring in-depth understanding of the underlying computational fluid dynamics principles. The automation process spans all simulation phases, starting with realistic fiber modeling via a custom script and followed by input specifications, automatic generation of the necessary geometry, export of the resulting computational mesh into ANSYS Fluent, control of computational solvers governing fluid flow and species mixing models, and ending with the automation of ANSYS CFD-Post to output solution data. This method is presented in a step-by-step manner through a case study of a Y-shaped microfluidic channel with an embedded electrospun nanofiber mat, the mixing performance of which is determined in this report's conclusion.

Contents

List of Figures

1 Introduction

1.1 Laminar Mixing

On the macro scale, turbulence is ubiquitous. However, as bio-analytical sensors and devices become increasingly miniaturized, the transition from a random, disordered flow regime into an environment with low Reynolds numbers and laminar flow is inescapable. Thus, techniques by which to accelerate diffusion-limited mixing have become a necessity to efficiently mix two or more fluid streams.

Publications detailing new geometric specifications for microfluidic channels that feature embedded mixing elements are very popular in the biosensor, lab-on-a-chip (LOC), and micro-total analysis systems (μTAS) fields, and typically feature an arrangement of one or more passive mixing elements from a wide assortment of baffles [16], serpentine or winding flow paths [17], split membranes [7], inlet- and channel-splitting arrangements [6], interdigital layers for multilamination [13]. Instead of passive elements, mixing may also be achieved through the use of active mixing techniques, which require external power, such as electrokinetic instability (EKI) components and pulsing micropumps [7].

These mixing enhancers typically function by shortening the mixing length separating different inlet solutions, in order to decrease the time scale necessary for diffusive mixing to occur. These methods typically display impressive mixing results or present ingenious microchannel designs; however, the increased complexity of the fabrication processes that are necessary to produce these precise patterns, 3D geometries, and intricate channel features add relatively high manufacturing costs, and possibly decrease the usefulness of the channels as pressure drops increase, and as less room remains available for functionalization or analytical uses [14].

1.2 Electrospun Nanofibers as Mixing Elements in Microfluidic Channels

Electrospinning is the process by which very thin strands of charged polymers, possessing diameters on the order of hundreds of nanometers, may be generated through the application of an electric field, which drives the layered deposition these thin nanofibers on top of a target area. Electrospinning has been demonstrated to be a cost-efficient and easy-to-implement solution for several types of separation processes, filtration, and for analyte capture [18, 9]. In this report, rather than attempt to develop an electrospun nanofiber system for biosensing purposes, an investigation is instead performed to determine the potential of nanofiber mats to function as passive mixing elements in microfluidic systems. A comprehensive approach is taken to produce a fully automated and configurable workflow by which the mixing produced by a nanofiber mat in a Y-shaped channel may be simulated using computational fluid dynamics.

2 An Algorithm for Generating 3D Fiber Systems for Simulation and Computational Analysis

In order for the necessary mixing simulations on nanofiber-mat models to be performed, and before any flow simulations could be performed, the initial requirement was to formulate a method by which fibers in a defined volume of any size could be generated. In this section, the relevant requirements of the fiber-generation system are discussed first, followed by the presentation and discussion of the resulting "sphere-stepping" algorithm that was implemented in Java to achieve this function. A brief comparison is then performed with fiber models created by the commercial software GeoDict, which, among several others, contains a specialized fiber-modeling module called FiberGeo. This module is intended to generate models of woven and nonwoven fibrous media, on which computational analyses that elucidate characteristics of the media–such as its permeability, pore size distribution, and filtration capacity–may be performed [21].

2.1 Scripting Requirements for Generating Model Fiber Systems

2.1.1 Uniformity

The most important requirement for generating of realistic fiber models is to emulate the natural uniformity in the geometric appearance that is visible in microscopy images of electrospun nanofiber mats. Despite their curvature and curling, the fiber mats are uniformly dispersed; there are no major variations in the density of the fibers, nor any tendency to exhibit large amounts of the knots-on-a-string appearance that is common in some nonwoven applications like filters imbued with binding agents, such as glues and resins, to maintain the fibers in a cohesive state [21]. Figure 1 compares two types of nonwoven fiber mats: the first is kept from disintegrating into individual strands through the addition of a binding agent, which holds together the nonwoven fiber mat. However, electrospun nanofibers more closely

resemble both the behavior and appearance of the *entangled* fibers shown in Figure 1(B), since they do not require any glue, and remain cohesive even against fluid flow without falling apart, due to the entanglement that occurs during spinning and the channel-edge anchoring that is achieved during hot-press channel fabrication. The fiber mats visualized in Figure 1, however, are much more densely packed, relative to their diameter, than the nanofibers utilized for the mixing experiments.

2.1.2 Customizability and code accessibility

Since the development of the fiber-generation software began before any computational analyses were performed, it was essential to clearly document the functions and methods in the script as well as write them as robustly and open-endedly as possible, so that improvements and modifications to the script and its resulting output could be efficiently performed. In addition, it was essential that the behavior of the script be straightforward to adjust, in order to allow other users without much programming experience to customize its actions and tailor its results to a specific problem, without having to navigate nor understand the entire Java implementation.

The output format of the resulting fiber data also required attention, since the parameters that required explicit inclusion in the output were thus necessary to calculate in the body of the script; if coordinate points were generated to define segments of the fibers, for example, it was necessary to know whether the output required the connection of subsequent points to be explicitly stated, or whether the points could simply be listed in order and assumed to be consecutive as they were read and imported.

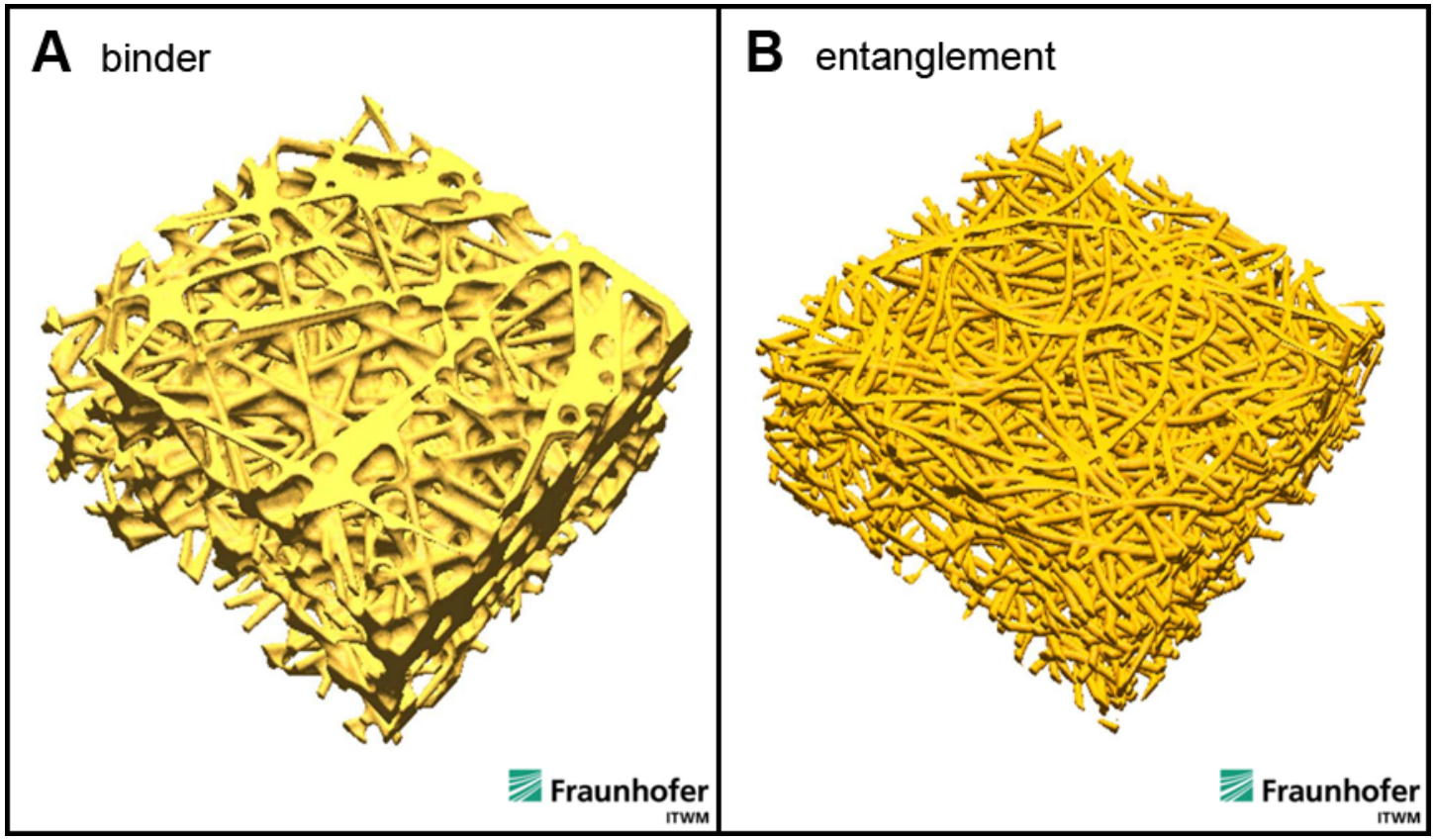

Figure 1: A comparison of two methods by which individual fibers in nonwoven fiber mats are held together. **(A)** The lumpy appearance of some nonwoven fibrous media is due to the use of a binding agent, in the form of a glue or resin, that holds individual fibers together and prevents tearing/separation of the individual fibers. **(B)** Nonwoven fibers possessing a high degree of curvature and display curling tendencies that allow individual fibers to become entangled during the spinning/extrusion process, thus keeping them bound without requiring the addition of any binding agent. *Source: Pore Scale Modeling of Porous Layers Used in Fuel Cells, by Jürgen Becker* [5].

2.2 Development of the Sphere-Stepping Algorithm

2.2.1 GeoDict's fiber-generation algorithm

Most algorithms for generating randomly elongating curves or lines, as required for the nanofiber script, function with an iterative elongation algorithm: in each iteration, a segment of some finite length is added to the "growing end" of the fiber, and this is repeated until the growing fiber fulfills an end condition (such as extending out of the pre-set boundaries of the defined fiber mat volume), and a termination action is performed (such as "slicing" the fiber at the control-volume edge) before a new fiber is initiated and elongated following the same method.

The algorithm implemented by GeoDict is based on equation 1 [5], shown below, which defines the direction of the growing fiber:

$$\left(\vec{d}_{n+1} - \vec{d}_n\right) - \left(\vec{d}_n - \vec{d}_{n-1}\right) = G_\sigma(0) - \alpha\left(\vec{d}_n - \vec{d}_{n-1}\right) - \beta\left(\vec{d}_n - \vec{d}\right) \tag{1}$$

in which:

$\vec{d}_{n+1}$ is the fiber direction, defining the direction in which the next segment's endpoint will be placed relative to the current endpoint,

$\vec{d}_n$ is the direction of the current endpoint relative to the second-to-last point,

$\vec{d}_{n-1}$ is the direction of the second-to-last point relative to the third-to-last point,

$\vec{d}$ is the direction of the full fiber relative to its start point,

α is the "straightness" of the fiber,

β is the "force" of the fiber, and

$G_\sigma(0)$ generates the randomness of the iterative point placement.

The contribution of "force," β, is used to scale the contribution of the vector $\vec{d_n} - \vec{d}$, which defines how much impulse is applied to push the growing fiber "back on track" relative to its overall direction. The contribution of straightness, α, is used to adjust how much the fiber deviates from local deviations in direction, since it affects the quantity of the vector $\vec{d_n} - \vec{d}_{n-1}$, which defines the impulse necessary to correct for the end point's deviation from the point before. Therefore, with $\alpha = 1$ the fiber grows in a straight path, whereas with $\alpha = 0$ there is no such local correction and curvature is maintained instead. With $\beta = 1$, the fiber maintains a consistent overall direction of travel despite small local variations; with $\beta = 0$ the fiber does *not* maintain an overall direction whatsoever, and instead maintains somewhat constant curvature. Lastly, if $\alpha = \beta = 1$, then equation 1 is reduced to simply $\vec{d}_{n+1} = G_\sigma(0) + \vec{d}$, which produces completely linear fibers in the direction $\vec{d}$.

2.2.2 Formatting requirements for ANSYS DesignModeler

When ANSYS Workbench 14.0 was chosen to be the modeling and simulation platform for all simulations, the required output format of the script was defined as being a list of coordinate points that could be imported to define a 3D Curve object, from a file that is formatted as shown in the following example:

curve ID	point ID	x -coordinate	y -coordinate	z -coordinate
1	1	$x_{1.1}$	$y_{1.1}$	$z_{1.1}$
1	2	$x_{1.2}$	$y_{1.2}$	$z_{1.2}$
1	3	$x_{1.3}$	$y_{1.3}$	$z_{1.3}$
2	1	$x_{2.1}$	$y_{2.1}$	$z_{2.1}$
2	2	$x_{2.2}$	$y_{2.2}$	$z_{2.2}$

...

In this format, multiple coordinate points are defined and assigned consecutively, and then imported into the native computer-aided design (CAD) utility in ANSYS. Curved splines are generated between successive coordinate points that possess the same curve ID, and thus the

points define the shape of each fiber. In the example above, two curves are generated, one defined by three and the other by two coordinate points. The coordinates of all of the points defined in the sample above must be inserted into the submatrix:

$$\begin{bmatrix} x_{1.1} & y_{1.1} & z_{1.1} \\ \vdots & \vdots & \vdots \\ x_{2.2} & y_{2.2} & z_{2.2} \end{bmatrix}$$

In order to generate the coordinates of the points that will be entered into such data files, as well as properly enumerate the curveID and pointID parameters of each point, a script was implemented in Java to output a text file containing these parameters and coordinates in the correct format. The algorithm utilized by this script proceeds conceptually as follows to generate the fiber models:

1. Dimensions of a 3D bounding box that defines the enclosed space in which the fibers will be generated are inputted by the user.

2. A point is generated at a randomly on one of the six faces of by the bounding box defined in step 1.

3. A virtual sphere is centered on this point, the radius of which is set by a user-defined value.

4. A point is chosen from the surface of the virtual sphere, and a curve segment is extended from the sphere's centerpoint to this new point.

5. The sphere is centered on the coordinates of the new point, defining a new surface from which the next point may be chosen.

6. Steps 4-5 are repeated iteratively, with a new point and curve segment being generated with each iteration, until a point is generated outside of the bounding box, which

signifies that the growing fiber successfully: (1) began its growth on the surface of the bounding box, (2) been elongated iteratively within the boundary zone to generate a random path, and then (3) exited the bounding box after being randomly elongated within.

7. Once the fiber elongation is terminated, the script chooses a new point on a random face of the boundary zone as per step 2, and enters another instance of the elongation loop to produce a new, separate fiber in the same manner.

8. New fibers are continuously generated until the end condition is reached, which is defined in this algorithm as the successful generation of the total number of separate fibers requested by the user. Once this occurs, all elongation/new-fiber iterations are stopped, and the coordinates of all the generated points are written to a data file along with their curveID and pointID, which is saved in a user-defined directory.

This algorithm is illustrated in Figure 2 on the next page.

2.3 Controlling the Script for Custom Fiber Control

2.3.1 Controlling directionality and curvature/curling

Options that alter the behavior of this algorithm may be toggled by simply adjusting the parameters that are presented in the script's source code. Adjustment of the script may used to customize attributes like the tendency of the fibers to elongate in one particular direction rather than randomly in the volume (by limiting the portion of the sphere surface on which the next point may be generated), as well as the curvature size of the fiber (by modifying the radius of the sphere; smaller radii produce closer coordinate points, resulting in increased curvature). For example, to make the fibers grow in a much more planar fashion, manner, the sphere may be flattened such that the fiber elongates primarily in the horizontal plane and only slightly in the vertical plane (see Figure 3(A)). Alternatively, to generate fibers that elongate primarily in one direction, similar to increasing GeoDict's "force" coefficient,

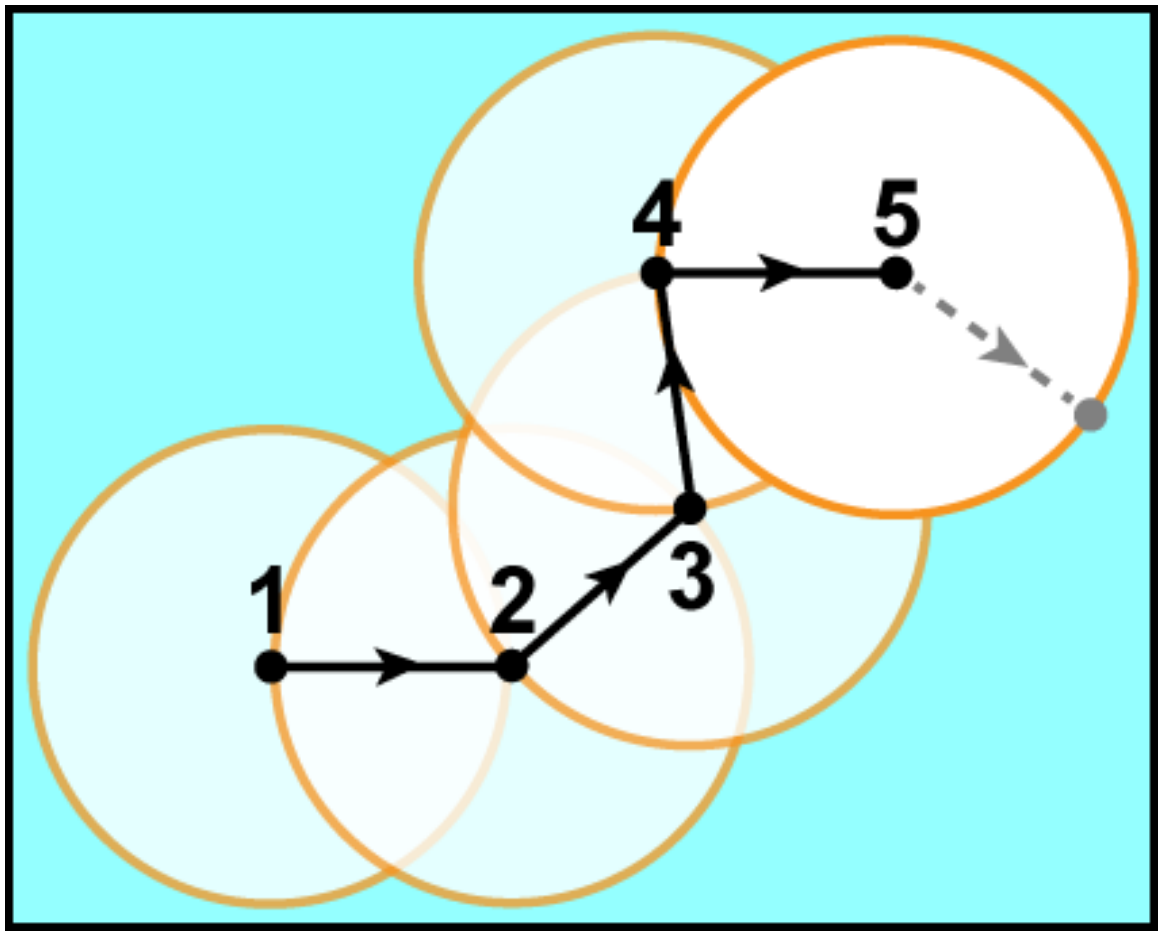

Figure 2: A 2D visual representation of the default behavior of the sphere-stepping algorithm for nanofiber modeling. To place each successive coordinate point in an elongating fiber, a virtual sphere is centered on each endpoint. A new coordinate point to which the fiber will elongate is then chosen from the surface of the sphere. After the point is chosen, the virtual sphere is centered on this new endpoint, thus defining the surface on which the *next* point will be generated. Repetition of this algorithm produces a fiber with random curvature, the extent of which is determined by the proximity of successive, randomly generated points.

the active surface zone on the sphere on which new points are generated may be limited to a smaller patch directly opposite the , determined by prior curvature or overall directionality (see Figure 3(B)). These modifications may also be used to directly steer the growth direction, as shown in Figure 3(C) and (D). To prevent fibers from exiting the fiber zone, the point-generation spheres can be set to increasingly "deactivate" the portions of their surface closest to an approaching boundary. This creates a repulsion effect, as the growing fibers become increasingly likely to choose successive points that face away from an approaching boundary. If applied gradually, this effect gently pushes growing fibers away from the regions through which they should not pass, while still maintaining a normal and natural appearance and distribution the fibers.

As an alternative to curved fibers, many publications that investigate fluid flow through fibrous media (which are almost entirely *permeability* studies on filtration media [19]) approximate these more intricate fibers with completely linear fibers. These linear fibers are very simple to generate with the script, by simply picking two random points on different faces and connecting them with a linear segment. This is repeated to produce as many linear fibers as requested by the user. Distribution of these linear fibers may be further controlled by adjusting the probabilities governing which faces are most/least likely to be chosen. For example, to generate transverse fibers, the top and bottom faces of a bounding box could be assigned probabilities of 0, which would result in the generation of fibers that only spanned from one vertical face of the box to another.

Lastly, this fiber-generation script is entirely independent from any other third-party resources, programs, or plugins since the code is fully original. Currently, modifications to the script settings to enable/disable various options are made by simply opening the source code in a text editor, and modifying the script as one pleases.

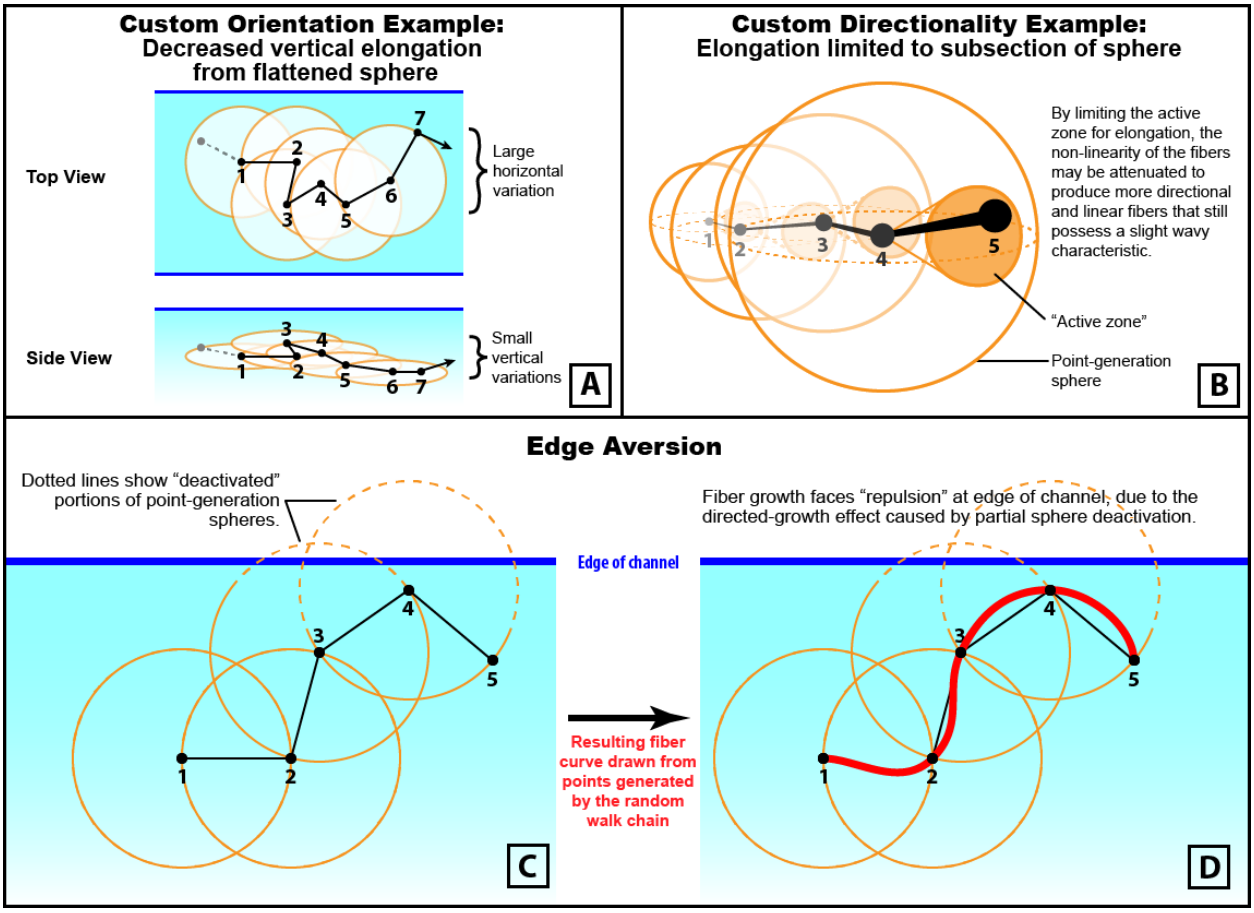

Figure 3: An illustration of various fiber features that may be generated through user modification of the sphere-stepping algorithm. **(A)** A method by which planar fibers may be generated, via with reduced variation in a chosen dimension. **(B)** A method by which relatively directional fibers, albeit with some local variation, may be generated; the virtual sphere's "active zone," which restricts the portion on which new points may be added, is reduced to a small portion of the total surface and oriented in the fiber's intended direction of growth. **(C)** A method by which elongation of the fibers may be directed away from incoming surfaces; by reducing the active zone through incremental "deactivation" of the portions of the virtual sphere closest to an approaching boundary, the growth of the nanofiber may be redirected away from the surface in a smooth manner. **(D)** An illustration of the actual appearance of the edge aversion shown in (C) on a spline defined by the coordinate points. The gradual redirection maintains smoothness in the fiber and does not result in behavior that looks unnatural or artificial. The appearance of this boundary-aversion method is shown in the rightmost pane of Figure 5.

2.4 Example Results and Comparison with GeoDict

The images in the following three subsections show the capabilities of the fiber-generation script, with comparisons to the results produced by GeoDict with equivalent settings enabled.

2.4.1 Linear Fibers

Linear fibers were generated with the modified version of the fiber coordinate-generator algorithm and displayed in ANSYS. The script was set utilize a cubic bounding zone with side length 1000 μm, and fibers with 10 μm cross-sections were generated. GeoDict was set to use a 500x500x500 voxel zone with a voxel length of 2 μm. The results from the ANSYS script and GeoDict are shown in Figure 4. The orientation, distribution, and appearance of the fibers were identical between the Java script and GeoDict's algorithm, the different color scheme notwithstanding. Following the 200-linear-fiber test, the number of fibers was doubled from 200 to 400, while maintaining the same boundary zone in order to verify that a higher fiber density continued to produce a similar output relative to GeoDict, and the test was equally successful, and also shown in Figure 4 on the following page.

2.4.2 Curved Fibers

Using the same bounding boxes as those from Figure 4, the curved-fiber features in the fiber generation script were utilized to create the three similar fiber mats shown in Figure 5 on page 34. The leftmost image in this figure contains 100 fibers in the bounding box, built with 300 μm spheres during the sphere-stepping algorithm, the center image has 300 fibers built with 500 μm spheres, and the rightmost image has 200 fibers built with 200 μm spheres and edge-aversion enabled for the fibers within the bounding box. The figure caption further details the characteristics of each variation.

As the spheres utilized for the random walk chain algorithm become smaller, the fibers curl and twist much more, since the points that control the curvature of the fibers are

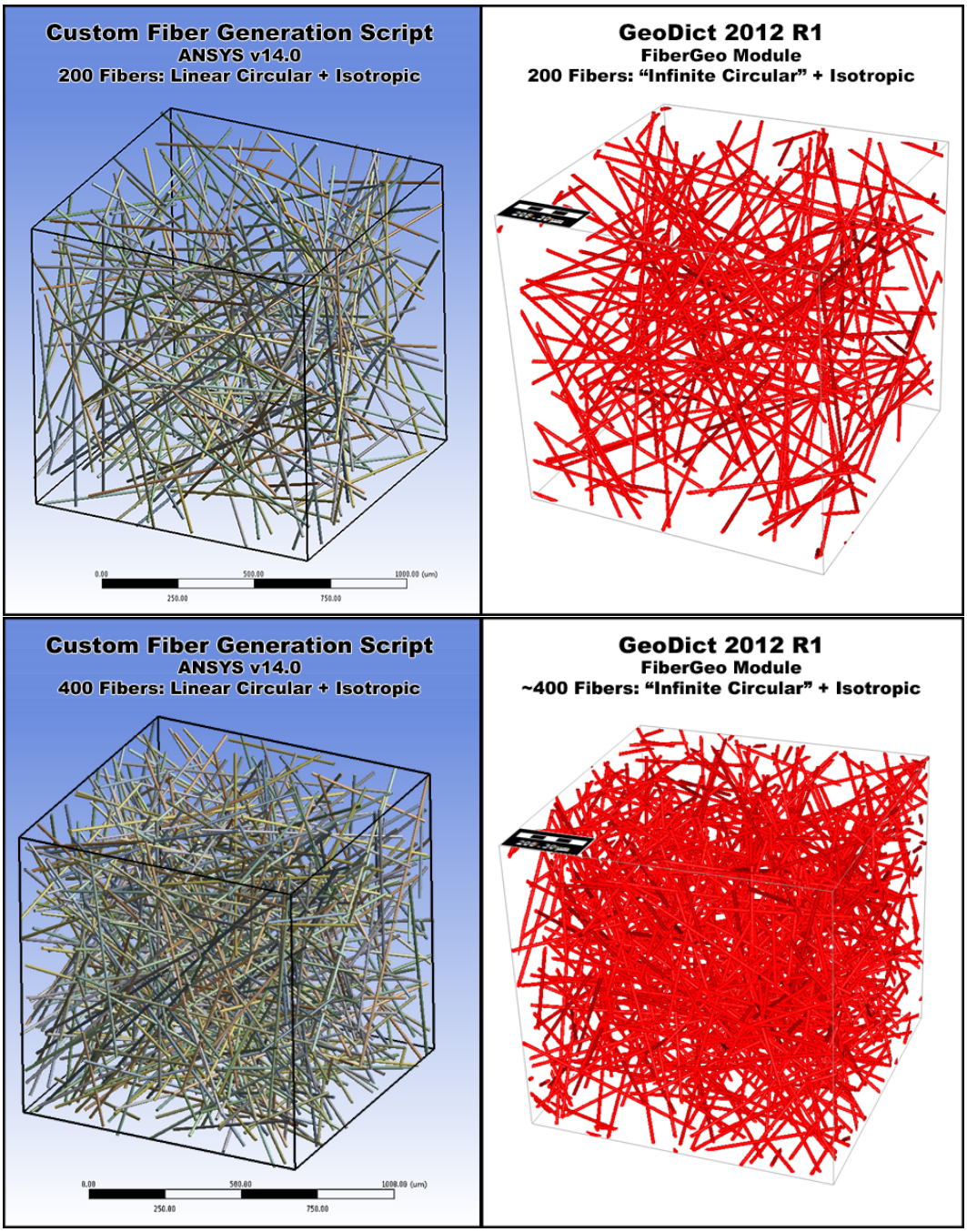

Figure 4: A comparison of the linear fiber models produced by the custom script developed for this report (visualized with ANSYS DesignModeler) with the output produced by GeoDict's FiberGeo module, when equivalent settings are applied. All fibers are generated with diameters of 10 μm in cubic regions with side lengths of 1000 μm. The custom script produces results identical to those generated by FiberGeo.

placed increasingly close together, resulting in increased variation within smaller scales. This effect is visible when comparing the very-highly-coiled fibers in the rightmost image in Figure 5(which used 200 μm random-walk spheres) with the relatively smooth and uncurved fibers shown in the center image (which used 500 μm spheres).

2.4.3 Comparison with nanofiber SEM imagery

Figure 6 on page 35 shows a comparison of the images captured through scanning electron microscopy with the results generated by the fiber-generation script (with an applied pseudo-SEM appearance), and highlights the realistic curved fibers that the script is capable of producing with the customizable, fast, and efficient sphere-stepping method.

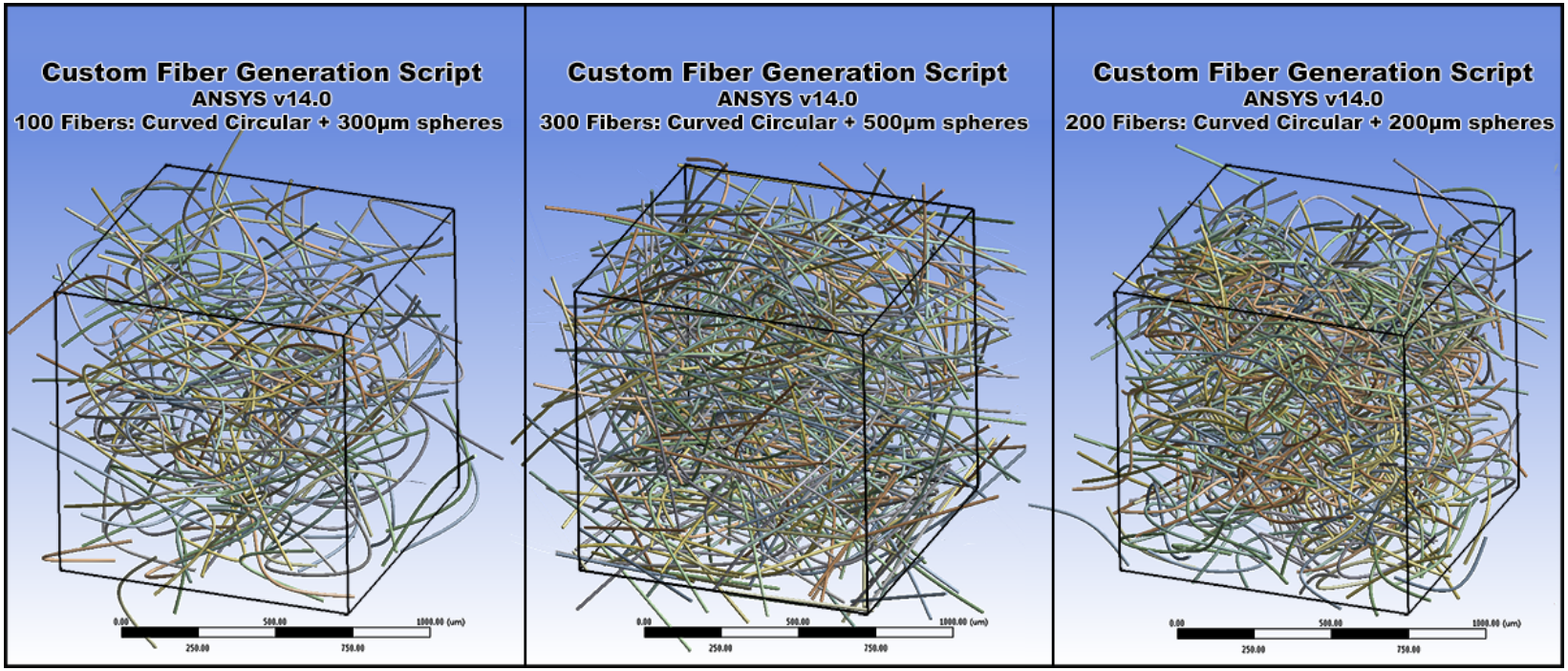

Figure 5: The result of three sets of curved fibers generated with the custom fiber-generation script, in which the model parameters were varied. The leftmost panel contains 100 fibers, the points of which were generated using spheres with radius 300 μm. The center panel shows 300 fibers generated using spheres with radius 500 μm, which are visibly less curved than either of the models by which the center panel is flanked. The rightmost panel contains 200 fibers that were generated using spheres with radius 200 μm, and for which the boundary-aversion feature explained in Figure 3(C) was enabled. The boundary aversion was automatically disabled once each fiber reached a threshold number of coordinate points, after which the aversion was disabled and the fibers allowed to exit the cubic boundary. Since the sphere radius was sufficiently large to skip across the regions in which the feature became active, some fibers exited the cube before attaining the threshold point-count. In addition, the effect of using a relatively small sphere radius is shown through the increased curvature of the fibers in the rightmost panel relative to the leftmost and center panels.

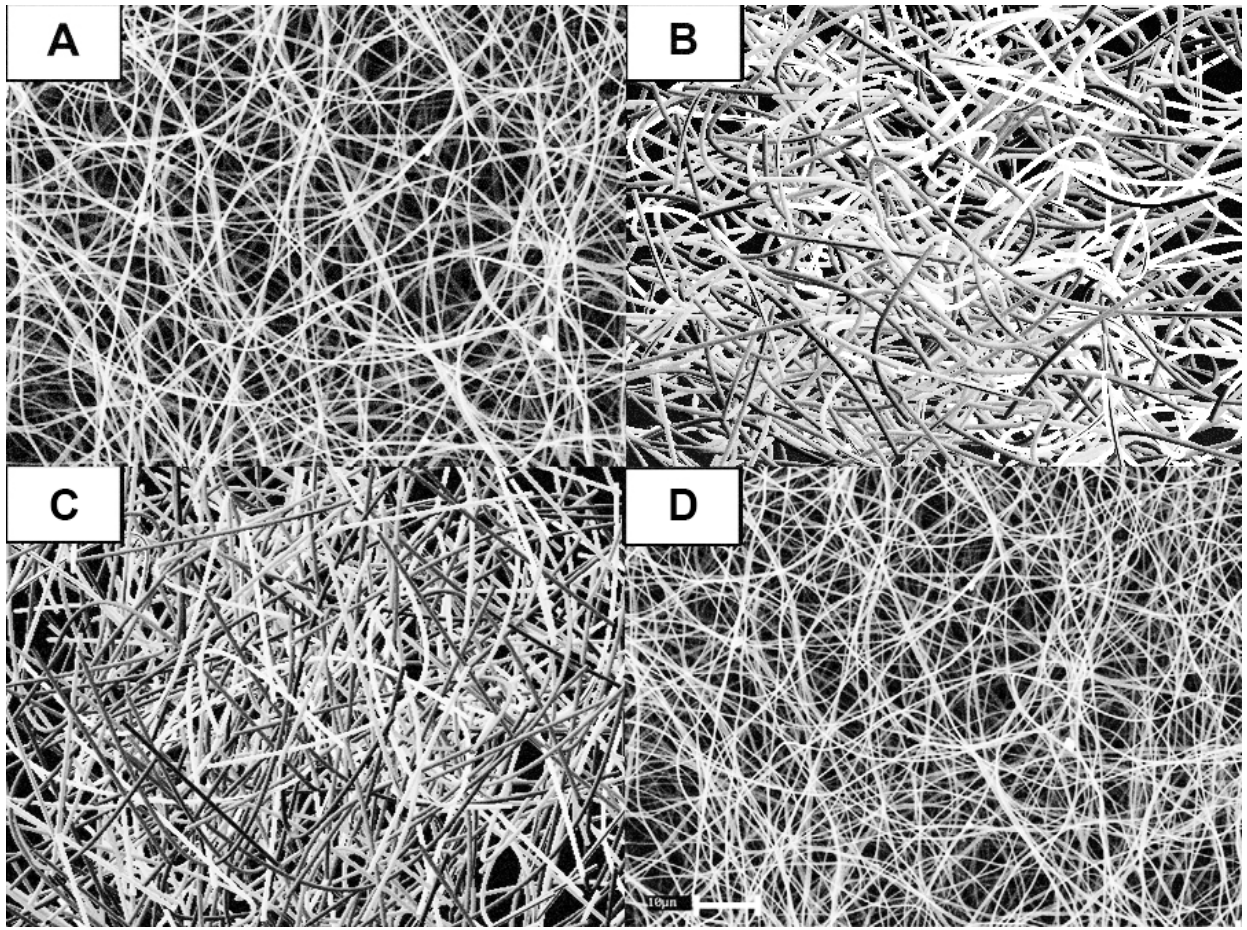

Figure 6: A comparison of the results produced by the fiber-generation script with real SEM images of nanofiber mats. SEM imagery is interspersed with the results from the fiber generation script to highlight the realistic results produced by the relatively simple sphere-stepping algorithm. Images (A) and (D) were captured via SEM, and images (B) and (C) were generated with the fiber generation script. Results (B) and (C) were captured with the same color scheme shown in Figures 4 on page 32 and 5 on the preceding page, converted to grayscale and inversely color mapped, after which their brightness was increased slightly using Adobe Photoshop in order to more closely simulate the appearance characteristic of SEM imagery. *SEM imagery provided by Lauren Matlock-Colangelo* [18, 9].

3 Modeling Electrospun-Nanofiber Mixing in Microfluidic Channels

3.1 Requirements for Developing the Simulation Workflow

The most important requirement of the simulation process was to develop a robust workflow that allows flexibility in all model inputs; the modeling scheme presented in this report is designed for easy adaptation to new channel specifications and to novel applications. As a result, although the model outlined in part 5.1 on page 52 is precisely specified, it may be very easily modified to accommodate changes in specifications that define either the dimensions of the base channel, the nanofiber arrangement/fiber diameter as well as the size and position of the mat, and the identities and concentrations of the modeled inlet species.

The next priority was to ensure that as many of the simulation phases as possible could be automated, while simultaneously providing sufficient user control of the automation. That is, a user who is not familiar with CFD and only slightly familiar with specifying CAD geometry should be able to follow the workflow outlined in this report and obtain accurate results that provide at least qualitatively accurate data (with which one could, for example, answer such questions as, "Will mixing performance be better when the fiber mat is placed close to the channel inlets, or closer to the outlet?"), with more expertise or experimental data required only to validate the model and fine-tune it for increased precision (e.g. a denser computational mesh for part 5.5 on page 60, or larger test volumes in part 5.3) and greater accuracy (e.g. consideration of species' adsorption to the nanofiber mats, the effects of which were assumed negligible for the theoretical species considered in this report).

It is important to note that the automation presented herein is not just a matter of minor convenience; several steps in the workflow require far too much repetition to be performed manually within any reasonable length of time. For these steps, automation becomes an essential component of the simulation without which they could not be accomplished, and not just an optional afterthought. First, generating the 3D fiber geometries in ANSYS De-

signModeler, each fiber possessing its own individually defined profile, is an extremely tedious task to perform manually since five separate operations (two points, a line, a plane, a sketch, and then a sweep) must be performed for every nanofiber in the model. Second, generating the large range of reference data for empty-channel diffusion profiles (see sections 5.3 on page 53 and 8.3 on page 97) require hundreds of very similar simulations to be processed in Fluent, and outlet contour plots need to be generated, for each profile, with identical camera settings and orientations. Without a method for automation, short-but-numerous tasks like these would prove incredibly tedious to perform manually.

3.2 ANSYS Workbench 14.0 with ANSYS Fluent Software for Modeling Fluid Dynamics and Dispersion

3.2.1 Overview of ANSYS Workbench 14.0

ANSYS Inc. develops engineering simulation software, and its newest iteration, ANSYS Workbench 14.0, is undoubtedly among the most feature-rich and well-supported of all commercial engineering simulation software packages. It owes its popularity in the simulation field to a design philosophy that integrates the advanced features demanded by industry experts with well-defined default behaviors, and an interactive graphical user interface (GUI) that is designed to make the use of these powerful features straightforward for new users and students.

The Workbench workflow begins with the CAD module called DesignModeler, in which parametric geometry is defined interactively with tools presented in the GUI. After the proper geometry has been generated, it is converted into a finite-element mesh on which finite-volume analysis may be performed for computational fluid dynamics (CFD) simulations. The finite volumes are generated with the native Meshing module, which automatically imports the parametric geometry created in DesignModeler and presents the user with several

overarching meshing schemes, each with their own algorithms and user inputs, with which the discretization of the finite volumes may be determined (optimal meshing parameters for the geometry utilized in this report are discussed in part 5.5 on page 60). ANSYS Fluent is then used to perform CFD simulations on the finite volume mesh, the results of which may be exported to either third-party applications or opened in ANSYS CFD-Post for further analysis and visualization.

3.2.2 ANSYS Fluent as the CFD solver

ANSYS Fluent provides users with a wide range of modeling capabilities, as well as the ability to perform multiphysics and multiphase simulations.
The following physical models are among the many supported in Fluent:

Viscous model: Laminar- and turbulent-flow simulations.

Energy: Heat and energy transfer, energy-driven fluid flow (convection, buoyancy, viscous heating), as well as chemical reactions.

Species: Species transport, reactions, and complex combustion models.

Discrete Phase: Simulation of discrete, dispersed particles through Lagrangian trajectory calculations

Multiphase: Interaction between multiple miscible/immiscible phases and between mixtures of different fluids, via volume-of-fluid and Eulerian models.

Fluent utilizes the finite-volume numerical method, by which the governing equations of the physical models chosen for the simulation are solved over discrete control volumes. The

discretization of these volumes is achieved during the meshing process, in which a complex geometry is algorithmically subdivided into smaller and simpler geometric primitives. For example, a two-dimensional surface may be meshed with a one or a combination of the following types of shapes: triangles, quadrilaterals, and occasionally higher-order polygons such as hexagons (see Figure 7, in which a 2D surface mesh is shown). Three-dimensionally, meshing may be performed with tetrahedral, hexahedral, prism, and polyhedral elements, all of which are may be visualized as being extrusions of the two-dimensional shapes listed above. In fact, many meshing algorithms employ this exact process; surfaces are is first meshed with 2D elements, after which the elements are extruded parallel to their surface normals (a step commonly termed inflation, which is a reference to the "ballooned" appearance of the expanded surface), followed by the addition of more elements to fill the remaining space. Meshing of the specific geometry to be analyzed in this report is discussed more thoroughly in part 5.5 on page 60.

Note that herein, the terms element and cell are used interchangeably to reference a discrete finite volume (a single "block" in the mesh), since Fluent's spacial discretization is cell-based (hence the term finite *volume*) and utilizes the centerpoints of cells, as well as cell-average values, in its algorithms. However, when discussing finite *element* algorithms, discretization is performed with respect to the *vertices* of the mesh (called nodes), and in that context the term "element" refers to these nodes instead of the volumetric cells that they surround.

3.2.3 Native scripting options in ANSYS DesignModeler

CAD functionality in the DesignModeler is a somewhat simplified alternative to more complex CAD packages like Autodesk SolidWorks and Dassault Systemes CATIA. Despite the typical flexibility of ANSYS Workbench platform, problems sometimes arise due to oversimplification: unlike most commercial CAD software, ANSYS DesignModeler neither possesses the capability to accept command-line inputs by the user, nor does it *officially* make available

Figure 7: A polygonal mesh used to simulate pressure distribution on an F1 car. *Source: ANSYS Fluent Features documentation* [1].

a sufficiently complex application programming interface (API) to permit full automation of its CAD features through scripts. Although it does provide a very useful feature in the ability to create macro-style recordings of users' interactively generated sketches, it dedicates only a minor portion of its documentation to explaining the remainder of its JavaScript-based Scriping API. Of the numerous operations and geometric primitives that may be created interactively in DesignModeler through the GUI, it provides rather limited instructions for implementing only the following narrow range of features through script-based commands:

- Points
- Lines from Points
- Surfaces from Lines
- Cross Sections
- New Parts (from multiple separate bodies)
- Planes
- Extrusions
- Revolutions

- Sweeps
- Skin Lofts

This list excludes features such as Named Selections, which are not only essential to the entire simulation workflow defined by ANSYS, but are also required to generate some instances of the elements *included* in the list above. Beyond just the basic functionality afforded by the ability to create these basic elements, the ideal API would incorporate many additional "getter" and "setter" methods to easily select objects, refer to objects' specific attributes, and set the object parameters listed in the Details pane. Since the process of manually creating the individual fibers in each 100-fiber model requires 45 minutes to a full hour, it was necessary to develop some efficient means to automate DesignModeler beyond just the simple commands presented in the Scripting API.

3.2.4 General architecture of ANSYS DesignModeler and JavaScript GUI-based automation

The general software architecture of applications in Workbench is shown in Figure 8. Elements in the GUI such as buttons, menus, and panels are arranged using HTML and labeled textually via information in XML files, which facilitates processes like localization, language support, and modification of the different groups of features made available to various classes of users. At the other end of the architecture, the low-level core algorithms (which perform the viewport rendering processes, handle the algorithms involved in manipulating the parametric bodies/features that define and transform the geometry, and so forth) are implemented in C++ libraries, and are inaccessible to users.

Since the HTML and XML files only provide instructions for *displaying* the GUI, and the C++ core logic consists of the relatively low-level functions of DesignModeler that only

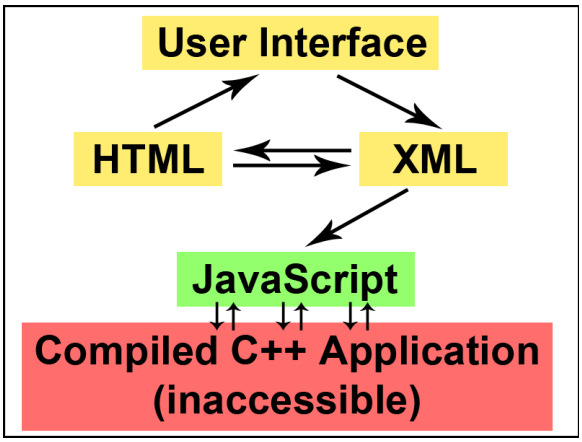

Figure 8: The overall model describing the architecture of ANSYS DesignModeler. *Adapted from "Application Architecture," by Matt Sutton.*

directly manipulate data, JavaScript is enlisted to interface these two otherwise fully separated layers by recognizing and relaying commands from the GUI elements to the core logic and back; it handles such tasks as mouse- and keyboard-input tracking, as well as event handling that occurs when buttons are clicked, menus accessed, and parameters modified.

By searching within the JavaScript source files (extension ".js") that are interspersed throughout various locations in the ANSYS installation directory and using word-of-interest search terms such as "Add," "Selection," and "Sweep," many references to additional, undocumented functions that are used by the JavaScript layer to call C++ methods were found, especially in the files named "agEventHandler.js" and "agInterop.js." The search terms "ag." and "ag.gui." in particular elucidated many invaluable functions necessary for developing an automation script, the usefulness of which was discovered after stumbling upon the following comment (which resides in both the "agMenu.js" and the "agInterop.js" files), which mentions the existence and purpose of commands prefixed with "ag.":

```
with (ag.wb.ScriptEngine)
{
    AddNamedItem ("ag", ag);  //Provides full access, but we do
                                not tell users about this
    AddNamedItem ("agb", ag.b);  //Batch command access
    AddNamedItem ("agc", ag.c);  //Batch constants access
    RunScript (scriptPath);
}
```

Searching yielded many useful, albeit undocumented, functions to supplement those presented in the Scripting API that is provided in the documentation. The two essential, unlisted functions necessary to complete the automation of the script are:

- `ag.gui.CreateSelectionSet();`

  This function generates a Named Selection, composed of all bodies in the active selection. It is necessary for the Path selection, if it is anything other than a sketch or plane axis, to be passed as a Named Selection to the Sweep operation. The only function defined in the Scripting API for line selection is the `GetEdge(i)` method of the LinePt object, but since this returns an Edge3D object and not the required Named Selection type, the batch command `agb.Sweep(...)` ignores this input.

- `ag.gui.ClearSelect();`

  This function clears all entities from the active selection, and is necessary to prevent previously selected objects from being unintentionally incorporated into a Named Selection generated using

  ```
  ag.gui.CreateSelectionSet();
  ```

Using these two commands, the following snippet of code can be written to generate linear fibers once line bodies connecting each pair of nanofiber endpoints have been generated (see part 8.2 on page 94 in the Appendix for the full script):

```
...
for (var j = 1; j < i; j++) {
    agb.ClearSelections();

    //Generate plane
    var fplane = agb.PlaneFromPointNormal(fp1.GetPoint(j, 1), LP[j].GetEdge(1));

    if (j < 10) {
        fplane.Name = planeName + "_0" + j;
    }
    else {
        fplane.Name = planeName + "_" + j;
    }
    agb.Regen();
    agb.SetActivePlane(fplane);

    //Generate circle sketch
    var ps2 = planeCircleSketch(new Object(), j);

    agb.Regen();
    //Generate sweep
    var sweep1 = agb.Sweep(agc.Slice, ps2.Sketch2, NS_Fibers[j], agc.AlignTangent,
                           1.0, 0, agc.No, 0.0, 0.0);
    sweep1.Name = "Sweep_" + j;
    agb.Regen();

}
```

The resulting commands in JavaScript that automate the geometry generation and operations are listed in part 8.2 on page 94, and is composed of recorded macros for sketch generation (functions planeSquareSketch(p) and planeCircleSketch(p, n) followed by an algorithm that works as follows:

1. A cube with 50 μm side lengths is generated by first calling the `planeSquareSketch(...)` method, followed by the `Extrude(...)` operation, with its depth parameter set to 50 μm in the direction normal to the XY Plane.

2. An `FPoint()` object is created and defined by the Coordinates from File method, in which pairs of coordinate points are imported from a text file to define the endpoints of an array of straight fibers (outputted by the custom fiber-generation script discussed in part 3).

3. A loop iterates through the index of the `FPoint` object, adding a new `LinePt` object with each iteration. To each `LinePt` object only a single edge is added, via the `AddSegment(Point, Point)` method, since storing multiple discrete segments within one

`LinePt` object caused issues with incorrect edges being returned by the `GetEdge(i)` method. Since only one segment is added to each `LinePt` object, it can be referenced with the function call `GetEdge(1)`without any chance producing errors or mis-selections.

4. A second loop is then performed with the purpose of adding the single edge in each `LinePt` object to its own Named Selection.

5. A third loop is then initiated in which a plane is generated at the base of each line via the `PlaneFromPointNormal(...)` function (using the first point in a coordinate pair and the edge between the points as parameters). After each plane is generated, a sketch of a circle that is 500-nm in diameter is generated at the origin of each plane using the `planeCircleSketch(...)` macro function, and then a Sweep feature is generated using the first point in the coordinate pair and the corresponding Named Selection containing the same edge used for the Plane.

This script is printed in the appendix, in part 8.2.

3.2.5 Automation of ANSYS Fluent and ANSYS CFD-Post

The process of automating both Fluent's CFD solver and CFD-Post's data output was, in contrast with DesignModeler, quite straightforward. This is due primarily to these applications' built-in journaling/macro-recording features, which allow all user GUI interactions or parameter modifications to be saved as a well-formatted list of commands. Elucidating the name and syntax of a command a simple task, which may be accomplished by simply recording a macro while performing the desired operations, and does not require searching an API or navigating through snippets of source code. The downside, however, is that macros from both of these modules utilize the IronPython programming language, adding yet another language with which a user must be at least superficially familiar to allow the customization of these macro-recording and -writing features.

Despite unavoidably being the fifth programming language required thus far in the workflow, Python is popular for its ease of use, and not much syntactical knowledge of the language is necessary to modify the macro files. In this report, automation was achieved in Fluent by toggling Workbench's "Record Journal..." command, launching Fluent and setting the necessary solver parameters, running the simulation, saving the computed results, and then entering Workbench once more and toggling the "Stop Recording Journal..." option to output a Workbench Journal file (extension ".wbjn"). As outlined in section 8.3 on page 97, a macro was also defined within Fluent in addition to the journal-based automation in order to maintain compatibility with the Case Modification dialog, as it automatically initializes the solver and subsequently modifies the case (by switching to the species transport model after sufficient iterations have been performed on the viscous flow equations to sufficiently converge their solution).

Automation in CFD-Post was performed in a similar fashion; a journal file was recorded while the desired functions were performed interactively on the first case, and then the recorded commands were modified slightly to iterate the same commands through subsequent cases. These commands are outlined in section 8.4 on page 101.

4 Error in CFD Models

Throughout every step of designing and executing a CFD solution, it is necessary to ensure that sources of error are controlled and identified in order to appropriately judge the value of the simulation results. This section outlines four main categories from which errors and inaccuracy arise in CFD simulations, and the methods by which these sources may be minimized.

4.1 Error Arising from Simplifications to Physical Models and Geometries

4.1.1 Balancing under- and over-simplification

When modeling physical phenomena, especially for CFD purposes, simplification is performed at the geometric and computational solver levels to ignore negligible contributions to the solution from minor details of the real system. Simplification reduces computational demand by limiting the complexity of the model. Geometrically, inessential features are often discarded, regardless of whether a CAD model is generated explicitly for CFD analysis or imported from an existing design. If the CAD file is external, and created with high detail for uses such as part specifications or mechanical engineering, the excess detail may be removed from the model through a process called defeaturing: in either a different CAD application or in ANSYS Meshing, the user defines a cutoff size under which to filter detail, and all elements flagged by the filter are removed, and all residual holes filled. For example, simplification is appropriate in aerodynamics simulations of passenger vehicles; the key slots, manufacturers' emblems, and tire grooves are defeatured from the CFD models, in order to avoid adding unnecessary complexities that do not significantly alter the final solution but add much unneeded complexity.

However, *over*-simplification by ignoring some relevant physics that produce a *significant*

effect on the solution introduces modeling error that reduces accuracy, easily to the point at which the reported data becomes completely irrelevant. Ignoring turbulence when modeling vehicle aerodynamics (which occur well within the high-Reynolds-number realm that indicates a dominantly turbulent flow) would produce an inaccurate model that does not at all describe the system for which it is intended. It is up to the user to extract the important contributions to the system from those that should not be modeled; CFD software rarely performs an incorrect calculation on a correct set of inputs, but it is more than willing to output a mathematically correct solution to an incorrectly defined case.

4.1.2 Geometric simplification

Geometric simplification is necessary in CFD analysis to maintain a sufficiently complex model while also managing the computational resources required to calculate a solution. Oversimplification of the geometry to the point at which it is missing relevant features from its real-world counterpart, however, will result in an inaccurate model regardless which parameters or schemes are enabled on the solver to enhance its accuracy. As with solver simplification, *geometric* simplification is highly dependent on the scope of the model; in single-vehicle aerodynamics models, representing the car with a large rectangular prism would yield useless results, but if the scope were enlarged to encompass wind flow patterns over a highway bridge, the blocky car representations would likely be more than sufficient.

4.2 Errors Arising from Discretization and Meshing

As discussed in part 3.2.2, when computing a CFD solution with the finite-volume method, the solver possesses than that provided by each element of the discretized mesh as a basis for the model's governing equations [15]. Finite-volume solvers function algorithmically by consideration of cell-center values and the gradients occurring both parallel and normal to the faces that comprise each element. Beyond this point, the local accuracy of the solution cannot be increased without further subdivision of the mesh; although higher-order

interpolation schemes may, for example, reconstruct linear gradients of quantities across cell faces [2], the detailed behavior of the fluid system cannot, by definition, be further resolved between the discrete elements with any more precision than that afforded by cell-average approximations or linear interpolation schemes. As a result, the construction of a sufficiently dense mesh to capture all relevant detail in the model is essential for a solution to have a stable convergence and an accurate result. Characteristic of an optimized mesh is that it contains sufficient closely packed cells in high-gradient zones that the relative gradient between cells is low, and that it the cells in relatively homogenous zones far from any gradient-defining boundaries are representatively larger to reduce computational requirements. For example, ANSYS generally recommend that "no flow passage should be represented by fewer than 5 cells" [3] in order to ensure that all near-wall boundary layers are captured, and their effects on fluid flow thus resolved properly in the calculated model.

The mesh may be manipulated to fulfill this requirement through a myriad of parameters, some of which allow control over cell structure, minimum and maximum edge lengths, the rate of cell-size growth as it expands from the proximity of a boundary, and so forth. For example, if the grid resolution of the mesh is doubled in each spatial dimension, discretization error is reduced by 50% on linear solvers and by 75% on second-order solver schemes [4]. Determining whether a mesh is sufficiently dense or properly structured is achieved through the investigation of mesh convergence, by which successively more detailed meshes are constructed and solved until there is no visible effect on the simulation outcome; when this threshold is achieved, the solution is deemed "independent of the mesh." In addition, Fluent also supports solution-adaptive mesh refinement, a process by which the solver dynamically adds elements to the mesh in regions at which the largest gradients of values between adjacent cells occur (and thus, likely targets the highest discretization errors caused by linear interpolation of non-linear phenomena) [3].

4.3 Errors from Incorrect Boundary Conditions:

Simply stated, if the simulation case is defined for the solver with inaccurate boundary conditions, the solution will lack accuracy. Because CFD analysis is typically performed to elucidate the internal behavior of a system when at least its inputs, if not also its observed outputs, are known, it would be impossible for a CFD simulation to produce an accurate solution if this information was incorrect or improperly inputted. At least some parameters such as inlet pressures, flow rates, material constants and identities, inlet parameters and locations, and slip or non-slip wall treatments must be defined before the solution may occur, and care must be taken to provide valid conditions; for example, when utilizing a pressure-based solver to model incompressible flow, setting the face-normal velocities on both the inlet and outlet is unnecessary–one is directly defined by the other through conservation of mass, and providing two separate values (especially if they are physically incompatible) only serves to over-constrain the model and yield solution errors and large inaccuracy in the output.

4.4 Numerical Error:

The last major class of errors involves those not directly attributable to any of the four other types of error mentioned above, but rather a result of the generation, manipulation, and storage of the solution data itself. These include such issues as insufficient iteration of the solver algorithm in order to achieve solution convergence (which are addressed by requesting more iterations of the solution from the solver), the explicit limitations in the precision of floating-point arithmetic (which may be overcome by utilizing double-precision variables), and communication errors with, for example, the MPI platform that enables parallel solution processing on multi-core/multi-node workstations and clusters. Typically, modern computer hardware is sufficiently robust that these errors are infrequent, unnoticeable, and typically do not affect the accuracy of solutions [11], although some manifestations may be easily visualized; the species model displayed in Figure 9 on the next page, for example, displays

a steady amount of fluctuating randomness as the size of its residuals (the calculated error in the model) converge to their precision limitation[1] relative to quantity being iteratively approximated.

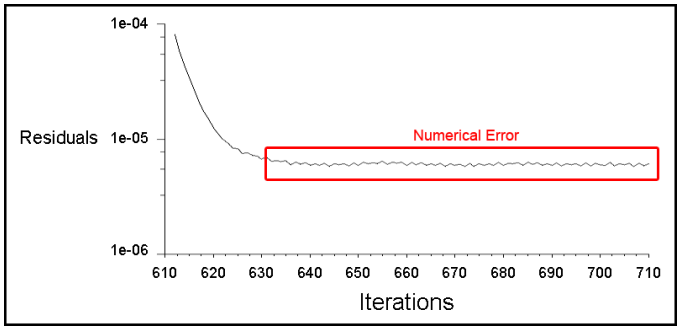

Figure 9: Numerical round-off errors visualized in the residuals plot of a species transport model as a solution was iterated in ANSYS Fluent.

[1]Although computer science authors do this explanation better justice, it is important to note that "maximum precision" does not impose relevant limits on the *order of magnitude* that these values may hold (except beyond a very, very large range), but rather on the size of the smallest finite difference between a rational number and its 32- or 64-bit floating-point representation; float- and double-type variables are stored internally by a means analogous to scientific notation, in which a series of digits–the length of which determines the precision of the floating-point–is multiplied by an implementation-dependent base raised to some power [12]. When base 10 is used, `floats` have a precision of six digits (before becoming prone to rounding errors) and the decimal point can be "floated" from 10^{-37} to 10^{37}. `Doubles` have a 10-digit precision, but the same exponential range. In practical terms, this means that while the numbers 12.3456 and 0.123456 can both be accurately defined by floats, (represented as 1.23456×10^{1} and 1.23456×10^{-1}, respectively), it would not be possible to maintain accuracy when computing their difference; the result would be 1.22222×10^{1} or 1.22223×10^{1}, depending on which rounding scheme is implemented.

5 Simulation Design, Physical Model Selection, and Appropriate Solver Schemes

5.1 Problem Outline

The workflow presented in this report is demonstrated through its application to a Y-shaped microchannel, in which the mixing of two solutions, one per inlet, is simulated. The two solutions flow into the channel, each through one of the two relatively short inlet channels (at the top of the double-pronged "Y"), which converge into a longer outlet channel in which which a nanofiber mat is placed. The purpose for the design of the model is to compare the mixing performance of the nanofiber-mat channel relative to the mixing that occurs due only to diffusion in an empty channel.

5.2 Geometry Specification and Boundary Conditions

The specifications of the channel's dimensions are shown in Figure 10 on the following page. The channel height is constant at 100 μm, the long outlet channel is 1000 μm in width and approximately 5 mm in length, and both of the inlet channels are 600 μm in width and 1765 μm in length (as measured by the length of their outside edge). The nanofiber mat was placed 540 μm downstream of the inlet junction, possesses a length of 2 mm, and occupies the full width and height of the channel.

The following boundary conditions were set for the the model:

- Velocity into each inlet channel is 0.00025 m/s, which corresponds to a total 1.5 μL/min flow rate through the channel.
- All walls of the channel and the surfaces of the nanofibers possess non-slip boundary conditions.
- Water, with no dissolved species, is flowed into the channel through the inlet on the right side of the channel (when the channel is viewed from the inlets in the downstream direction; the topmost inlet in Figure 10 is the "right side" inlet).

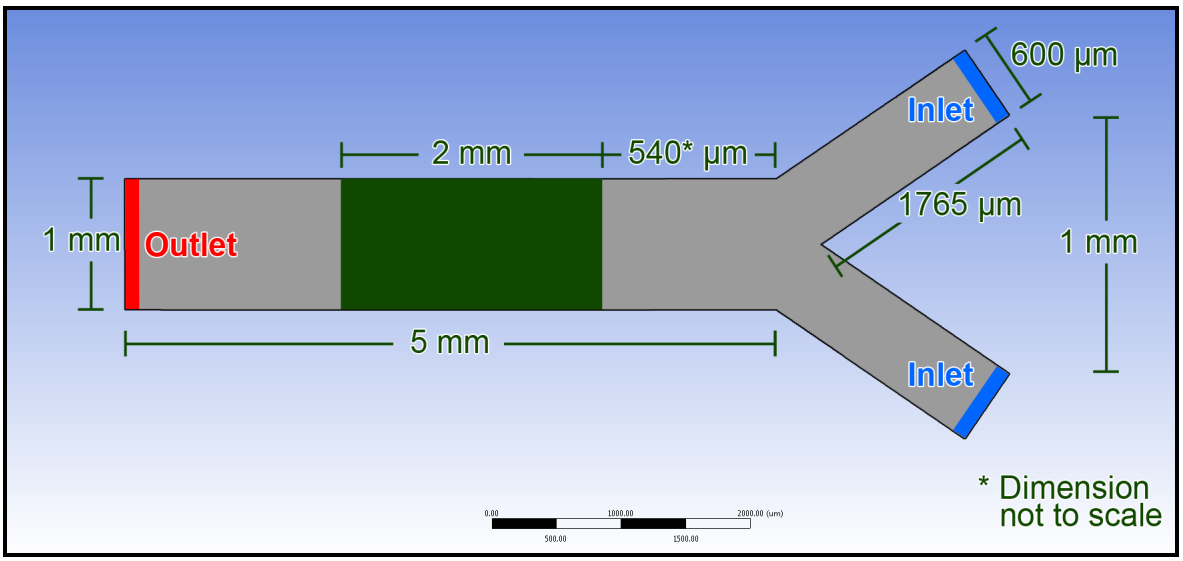

Figure 10: The geometric specifications of the microchannel in which the flow simulation will be performed. The green rectangle represents the area in which the nanofiber mat is placed

- Water containing a theoretical dissolved species dissolved, is flowed into the channel through the other inlet (i.e. the bottommost inlet in Figure 10).

5.3 Analyzing Nanofiber-Induced Dispersion: The Unit Cube

A vast disparity in scale that exists between the nanofibers, which are only 500 nm in diameter, and channel into which they are placed, which is two thousand of times wider and two hundred times taller than an individual fiber. This presents a large problem, in that both the nanofibers and the channel cannot be simultaneously represented as explicitly defined geometry in a solver model; if the scope of the model was sufficiently large that it encompassed the entire channel, modeling the fiber mesh geometrically would require far too dense of a mesh. If such a mesh were attempted with hexahedral elements possessing a mean edge length of 1 μm, 70,000 elements would already be required to simply span the channel's cross-section with a single-element layer, and 7×10^7 elements would be generated at every 1-mm increment in depth. Currently, to generate a solution for a channel-sized volume at such a high mesh density would require a powerful high-performance-computing environment, if not dedicated time on a supercomputer.

As a result, since it is impossible to model the individual fibers of which the entire fiber mat is composed, an alternate approach is considered to determine the mixing effects produced by the nanofiber mat.

5.3.1 Fick's first law and the dispersion model

Fick's law, which models diffusive flux in steady-state conditions, is written below in equation 2:

$$J = -D\nabla c \tag{2}$$

where

J is the diffusive flux of the species, with units of $\left[\frac{kg}{m^2 \cdot s}\right]$,

D is the diffusion coefficient, a property of the species that determines the speed of diffusion, with units of $\left[\frac{m^2}{s}\right]$, and

c is the species concentration, with units of $\left[\frac{kg}{m^3}\right]$, where ∇c is its generalized spatial gradient.

The Fickian diffusion scheme is used by default in Fluent when measuring species' diffusive transport, and was thus used in this report to model the diffusion of species both up- and downstream of the nanofiber mat. Within the mat, however, a modification to this method had to be performed in order to account for the dispersive fluid flow created by the nanofibers, which enhances species mixing. Thankfully, closely related to the Fickian diffusion scheme is the analysis of dispersion, which allows the enhanced mixing rate caused by a homogenous dispersive media to be modeled. The model itself consists of a simple adjustment to equation 2: a species' diffusion coefficient in Fick's Law, D, is replaced with the larger dispersion coefficient, E, which is defined by the value necessary for Fick's Law to reproduce the mixing profile/net mass transport contributed by both dispersive fluid flow through a volume of fibrous media, as well as the diffusion from from a species' inherent diffusion rate (see equation 3). Thus, E replaces a species' *native* diffusion coefficient, D, in

order to model the species' enhanced diffusive/dispersive transport through porous/fibrous media, by just scaling the diffusion coefficient of the Fickian model into lumped parameter that incorporates both diffusion *and* dispersion.

$$J = -E\nabla c \tag{3}$$

The dispersion coefficient is thus strictly neither a species property, nor a property of the media through which the flow passes, but a property of both entities simultaneously (it describes a specific species flowing through a specific media), and is thus valid only *within* the porous/fibrous media; as soon as the dissolved species exits the boundaries of the media, its diffusion coefficient must immediately be reverted to its species-specific value, D, as it was before the species entered the dispersive-mixing environment in which dispersion is modeled with E.

Porous media is an exemplary substrate for which dispersion may be approximated in this manner, because it is uniform, generally random, and possesses definite boundaries that separate the dispersion and Fickian diffusion models. The nanofiber mats fit this criteria exceedingly well; they are relatively homogenous throughout the channel in respect to porosity/density and fiber orientation/appearance, and thus their dispersive effects were also assumed to be homogenous within the boundaries of the nanofiber mat.

The approach used very frequently in the domain of fiber science and in the filtration/paper industry is to extract sample cubic sections from their porous media, analyze and quantify the cubic sections' dispersive behavior, and then extrapolate this behavior in a general fashion to the full porous zone by assuming that the sample represents the characteristic dispersion generated in the entire geometry [21]. In fact, this is the model for which software packages like GeoDict are designed (see Figure 4 on page 32 in part 5.3); GeoDict is used industrially by researchers to predict and analyze the characteristics of porous media and woven/nonwoven fibers by quantifying parameters like permeability, and filter capture efficiency in sample volumes that are assumed to be representative of the full domain from

which they were modeled.

For this report, the same approach was utilized to describe the enhanced mixing effects produced by the nanofiber mats via dispersion. The process of extracting a cubic section of nanofibers from the mat to determine its dispersive characteristics was performed by generating, through the automated methods that control fiber generation and ANSYS DesignModeler (discussed in parts 3 on page 36 and 3.2.3 on page 39, respectively), several linear fiber distributions in a relatively small cubic volume with edge length 50 μm. After the models were generated, they were meshed (see section 5.5 on page 60), and then analyzed in Fluent to define the appropriate dispersion coefficient that models their effects most accurately.

5.3.2 Data acquisition for fiber unit-cube models

An example of one such "extracted" nanofiber unit-cube model is shown in Figure 11 on page 58, in which the dispersive mixing effects by the randomly generated linear nanofibers were assumed to be characteristic of the entire nanofiber mat. In order to reduce the sensitivity of the calculated dispersion coefficient to specific geometric variations in the fiber unit-cube models, 9 sample cubes–each of which possessed a randomly generated internal fiber structure–were analyzed and their dispersive effects averaged to more accurately define the overall behavior of the entire nanofiber mat. Two opposite faces in each cube were chosen; one was assigned to be the unit cube's outlet, and the opposite face was split into two equal halves, each of which was defined as either the pure-water inlet or the species inlet (through which water with a fixed concentration of dissolved species would flow). In Figure 11 on page 58, the outlet is shown in green, the species inlet shown in red, and the pure-water inlet is shown in blue. The flow passes through the extracted portion of nanofibers on its journey from the inlets to the outlet, and by capturing the concentration profile of the dissolved species at the outlet, the lateral species transport generated by the nanofibers' dispersive effects can be measured through analysis of the outlet's species-concentration contour map

on the outlet surface. All deviation at the outlet relative to a completely unmixed, side-by-side inlet profile is quantified in this manner by simply performing a summation of the pixel-level absolute differences between an inlet contour profile and an outlet contour profile. This calculation is illustrated on a contour plot in Figure 12 on page 59, and additional sample results of the absolute-difference method are shown in Figure 24 on page 79.

Dispersion in the fiber mats was thus quantified by measuring to what extent the diffusion coefficient of a species would have to be increased to produce the same extent of lateral species transport, as produced by dispersion in the fiber unit-cube models. Contour plots of the outlet surface, colored with a linear grayscale gradient (pixel intensities mapped using the boundaries:intensity $0 \longrightarrow 0\ \frac{units}{L}$ and intensity $256 \longrightarrow 0.5\ \frac{units}{L}$), were assigned such that the highest concentration, specified as $0.5\ \frac{units}{L}$ at the species inlet, would be appear white and the lowest, specified as $0\ \frac{units}{L}$ at the water inlet, would appear black. Figure 23 shows sample outputs of the raw diffusion data, in which values for $D = 6.0 \times 10^{-11}\frac{m^2}{\mathrm{s}}$ through $D = 49.0 \times 10^{-11}\frac{m^2}{\mathrm{s}}$ with intervals of $0.5 \times 10^{-11}\frac{\mathrm{m}^2}{\mathrm{s}}$ are visualized. The data was gathered using the automation method for Fluent and CFD-Post explained in part 3.2.5 on page 45, with cubes consisting of approximately 200,000 elements.

5.3.3 Data acquisition for dispersion comparisons: fiberless unit-cube models

As stated in equation 3 on page 55, the dispersion-model adjustment to Fick's law requires that a lumped dispersion coefficient be elucidated, and exchanged with a species' native diffusion coefficient to produce an equal amount of lateral species transfer in an *unobstructed* volume as the species would otherwise undergo when flowing through an the same volume of *porous* media with its *original* diffusion coefficient. Thus, to create a range of standard values against which the fiber cubes' profiles could be fit, species transport was also simulated inside empty, fiberless cubes of the same dimensions and parameters as those *with* fibers, except across a large range of applied diffusion coefficients. The outlet contours resulting from the capture of these diffusion-governed concentration profiles in the fiberless cubes

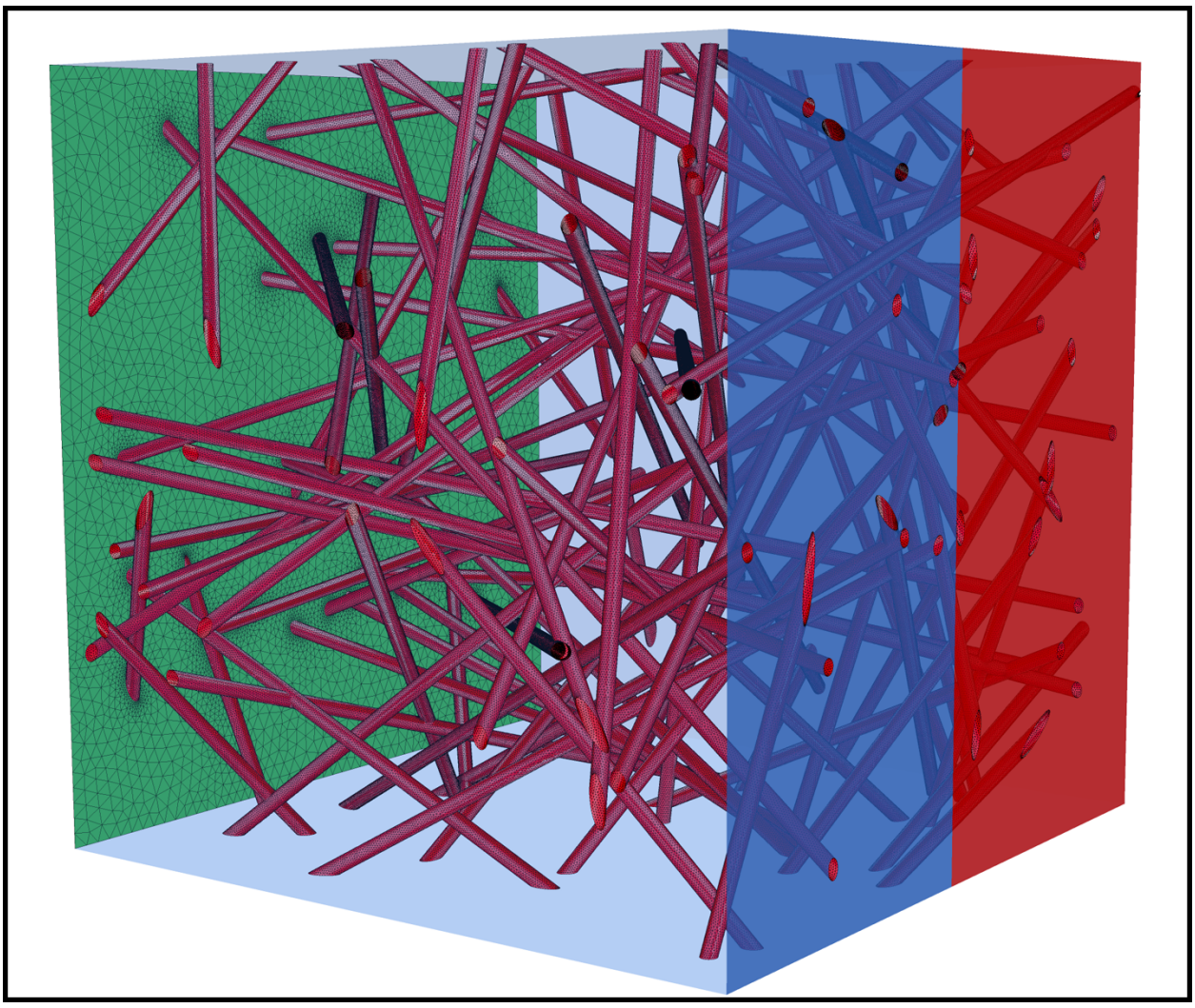

Figure 11: An example showing a unit cube with 100 randomly generated nanofibers passing through its interior. The inlets of the unit cube are shown in blue and red, and the outlet is shown in green. Pure water flows into the cube through the inlet highlighted by the blue face, and water with 0.5 $\frac{units}{L}$ of a theoretical dissolved species flows through the inlet highlighted by the red face. If the diffusion coefficient of the dissolved species is set sufficiently low, (e.g. $D = 1 \times 10^{-11} \frac{m^2}{s}$ as used in the fiber unit-cube models), any changes in the sharp, vertical concentration profile of the dissolved species at the inlet, as the fluid flows towards the outlet, may be attributed to dispersive effects produced by the modeled nanofibers.

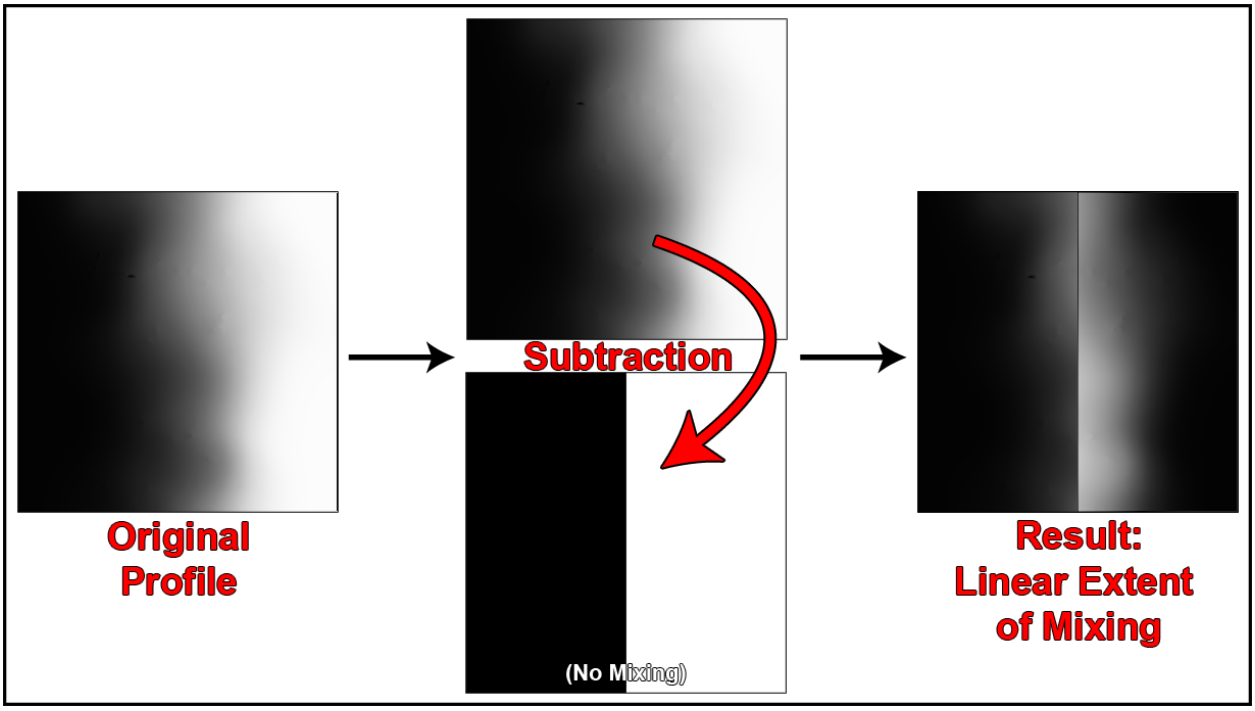

Figure 12: An illustration of the absolute-difference method, by which the change in species concentration between the inlet and outlet is calculated. Since the solute concentration in the contours is generated with a linear grayscale intensity map, the absolute difference between the outlet profile (top center) and the inlet profile (bottom center) yields the absolute change in species concentration at each pixel in the images, while also retaining the *same* grayscale-map scale from the original image (since no actions were performed except subtraction, which is linear).

were generated for the ranges $D = 1.5 \times 10^{-11} \frac{m^2}{s}$ through $D = 6.0 \times 10^{-11} \frac{m^2}{s}$ in intervals of $0.1 \times 10^{-11} \frac{m^2}{s}$, and $D = 6.5 \times 10^{-11} \frac{m^2}{s}$ through $D = 50.0 \times 10^{-11} \frac{m^2}{s}$ in intervals of $0.5 \times 10^{-11} \frac{m^2}{s}$. After these 132 profiles were captured, they were processed in an identical fashion to the fiber-cube models, using the absolute-difference method outlined in Figure 12 to quantify the amount of lateral species transport that had occurred.

5.4 Calculating the Dispersion Coefficient of Fiber Unit-Cube Models

The dispersion coefficient was defined in equation 3 as the value to which a species' diffusion coefficient, D, must be increased in order for a Fickian diffusion model to reproduce most

accurately the dispersive mixing effects created by a porous/fibrous volume. Thus, all that remains in defining the dispersion coefficient for the mixing model is to fit the fiber unit-cube data to values gathered from the large range of possibilities in the fiberless unit-cube dispersion data, and to identify which elevated diffusion profile produced equal amounts of lateral species transfer as the dispersive fiber unit-cube models.

To accomplish this task, least-squares data fitting was performed. First, a summation of the squared intensity values of each pixel in the absolute-difference contour plots (discussed in part 5.3.3 on page 57) was performed, in the image-processing application ImageJ, and the results stored in an Microsoft Excel spreadsheet. Then, the absolute difference between the sums resulting from each of the 9 unit-cube fiber models was plotted against each of the fiber-free diffusion cubes' sums. The fiberless diffusion-cube profile with which each fiber unit-cube model possessed the smallest absolute difference was identified as being the most representative dispersion coefficient with which the dispersive species transport in the fiber unit-cube model could be modeled (see Figure 26).

5.5 Creating and Meshing the Unit Cube Models

The unit cubes were meshed in ANSYS Meshing using tetrahedral cells, generally resulting in meshes composed of 3.5×10^6 to 6×10^6 finite volume elements, depending on the fiber-geometry-specific factors inherent to the specific fiber geometries in each cube. For example, if the fibers were randomly generated such that several were in close proximity for extended lengths, many small finite-volume cells would be required to define the boundary layers present between the two fiber surfaces. Mesh convergence tests, as discussed in part 4.2, were performed using three meshes of varying density that were generated using the same geometric unit-cube model, ranging from a low to a high extent of mesh refinement with the following number of elements in each mesh: 1.2×10^6 elements in the low-density mesh, 4.2×10^6 elements in the medium-density mesh, and 13.1×10^6 elements in the high-density

mesh. Differences between the models were quantified as percent values for both the diffusion profiles and outlet velocity profiles of the meshes.

Mesh independence was not observed visually during the transition from the medium-density mesh to the high-density mesh, since the outlet profiles appeared different (a much smoother solution was provided by the high-density mesh), but the data lifted from analysis of the contour plots generated by the two models varied by only 10%. Since the variations caused by slight mesh dependence were thus much smaller than the differences resulting from *geometric* dependence (the variations between which which could be averaged), and since increasing the mesh density above $\sim 9 \times 10^6$ elements resulted in large increase in the time required to converge a solution, the target mesh density was chosen to be in the neighborhood of 4.2×10^6 elements. Each mesh possessed an average orthogonal quality–a ratio of the smallest and largest distances between opposite faces in each cell[2]–greater than 0.85, which indicated a relatively good-quality mesh (especially considering the random geometry, which was not ideal for meshing algorithms by any means). If a mesh scored below 0.85 in this metric, parameters such as the curvature angle were modified slightly until the orthogonal quality increased beyond this cutoff value. Typically, viewport isolation of only the lowest-quality elements showed that these were located mostly in the boundary layers of the fibers, especially in thin gaps between *adjacent* fibers, and thus no major issues were raised because these boundary-layer elements were intended to be relatively thin in order to model boundary-layer effects sufficiently well. As an example, of the mesh appearance, the surface mesh of the 13-million-element fiber unit-cube model is shown in Figure 13. To contrast, the much more ordered mesh that is generated on the empty, fiberless diffusion

[2] Meshes possessing larger values of orthogonal quality result in faster solution convergence than meshes with lower values. As mentioned when discussing the residual equations in part 5.6.2 for convergence monitoring, each cell in the mesh is sensitive to adjacent cells with which it shares faces due to the conservation of equations applied to each side. If the a cell contains only a small volume but possesses a large surface area (e.g. a hexahedron with a side profile similar to a nearly-flattened parallelogram or a tetrahedron with one side very large relative to the others), the sensitivity of the cell to these external factors relative to its own cell values is very high. The more of these cells exist in the model, the less stable nearby cells behave, and the longer the convergence process requires.

cubes is shown in Figure 14 on page 64.

5.6 Solution Models and Convergence

5.6.1 Solution models

The Reynolds number, a dimensionless quantity representing the ratio between inertial forces to viscous forces in the fluid, is used to predict whether a flow is likely to be laminar, turbulent, or an intermediate transition. For porous media with cylindrical obstructions, the Reynolds number may be calculated as follows [8]:

$$Re = \frac{\rho v d}{\mu} \tag{4}$$

where:

Re is the Reynolds number,

ρ is the density of the fluid in units of $\left[\frac{\text{kg}}{m^3}\right]$,

v is the mean fluid velocity in units of $\left[\frac{\text{m}}{\text{s}}\right]$,

d is the diameter of the cylindrical fibers in [m], and

μ is the viscosity of the fluid in units of $\left[\frac{\text{kg}}{\text{m}\cdot\text{s}}\right]$.

Assuming the fluid possesses the material properties of water, $Re = \frac{\left(1000\,\frac{\text{kg}}{m^3}\right)\left(0.00025\,\frac{\text{m}}{\text{s}}\right)\left(5\times10^{-7}\text{m}\right)}{\left(0.001003\,\frac{\text{kg}}{\text{m}\cdot\text{s}}\right)} = 1.246 \times 10^{-4}$, which is far below the literature cutoff value of $Re = 0.1$, averaged over the entire media, over which the Darcy porous-media transport regimes give way to higher-order phenomena and dispersion of laminar flows through porous media begin to possess turbulent-dispersion characteristics [10, 8, 20]. Since it would require a one-thousandfold increase in the Reynolds number (e.g. a fluid velocity of 0.25 $\frac{m}{s}$ through the outlet channel in which the nanofiber mat is placed, which translates to flow rate of $1,500\ \frac{\mu L}{min}$) to attain this cutoff, the laminar-flow model is suitable for this scenario. In Fluent, the laminar scheme is enabled within the Viscous model, and boundary conditions are imposed such that every side of the channel and all fiber surfaces are assigned a zero-slip $\left(v = 0\,\frac{\text{m}}{\text{s}}\right)$ condition. In the Y-shaped

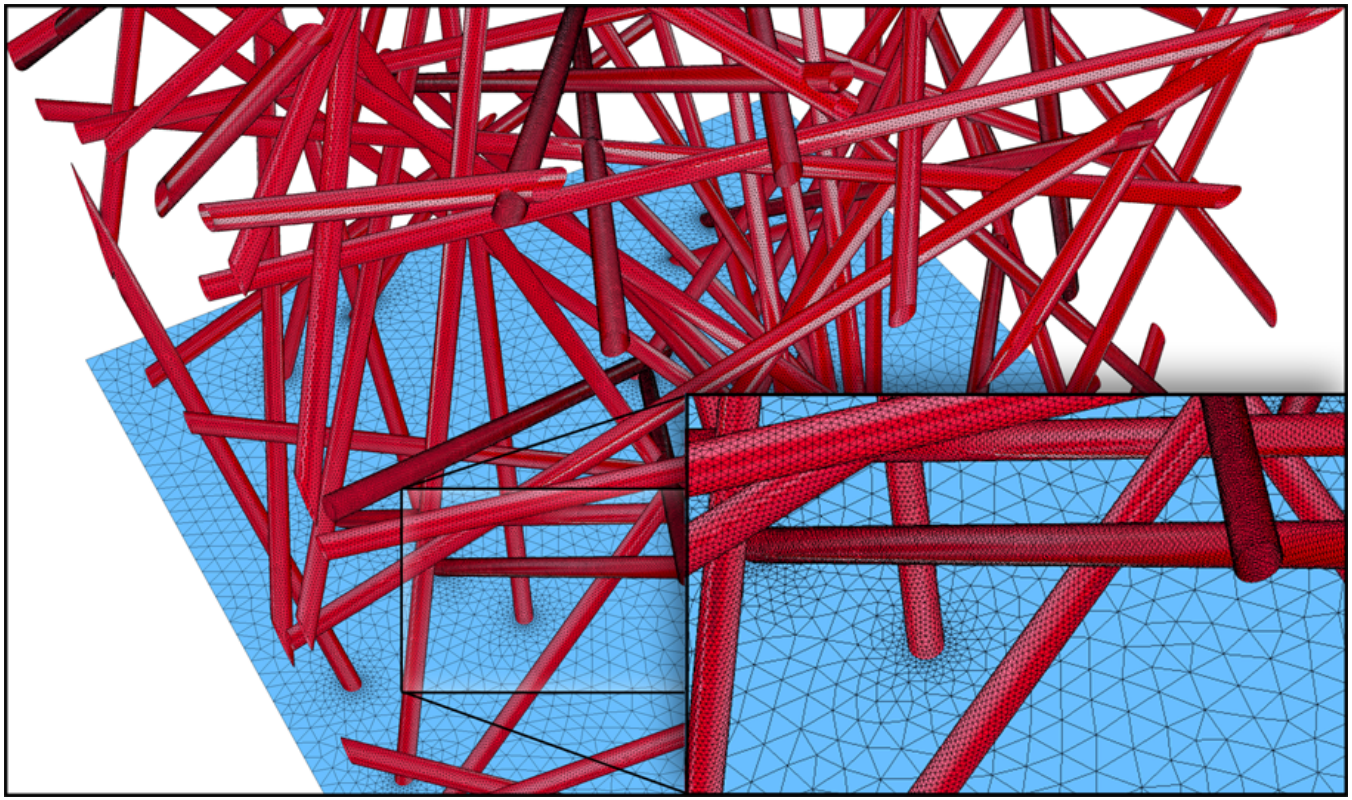

Figure 13: An image showing the fiber distribution in the cube, with all sides of the cube hidden except the outlet (which is shown in light blue). The inset provides a closer look at the boundary layers that surround each fiber, visualized on the surface mesh shown on the outlet face. The dense surface-layer mesh may also be seen on the individual fibers, and the dense tetrahedral boundary layers surrounding each fiber appear at the points through which the fibers pass through the side of the cube.

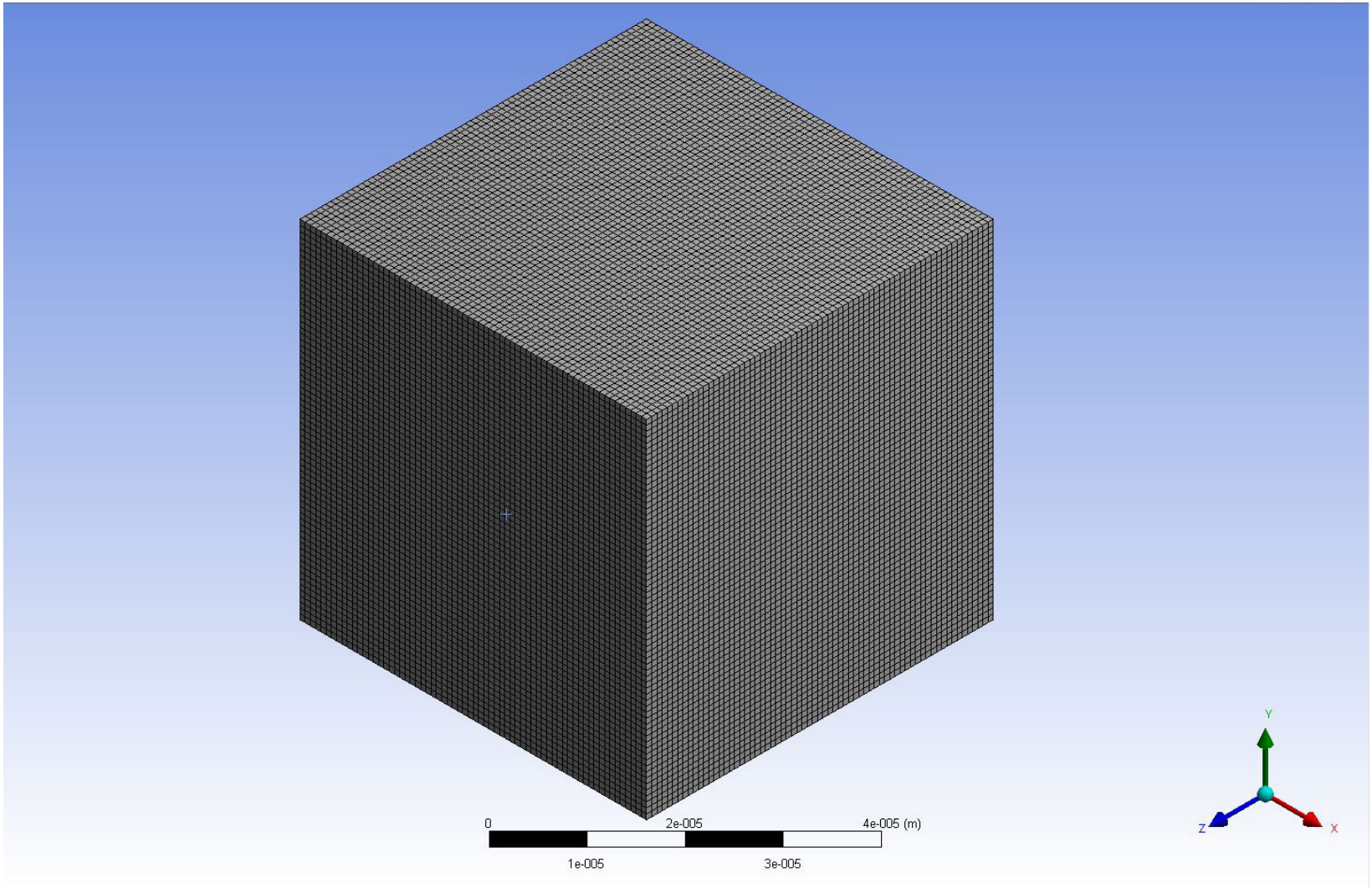

Figure 14: The density of the mesh used for simulations of empty unit cubes, the concentration profiles of which were generated for large ranges of diffusion coefficients and quantified as explained in part 5.3.3 on page 57. Since the smoothness and display quality of contour plots is affected by the mesh density, and since 200,000-element models (especially those that possess, by definition, perfect orthogonal quality and skewness metrics like this one) are relatively quick to solve, the automated script that loaded and simulated each case completed its task overnight.

channel, both inlets are assigned a velocity of $0.0002084\,\frac{\mathrm{m}}{\mathrm{s}}$ in order to achieve the desired average fluid velocity of $0.00025\,\frac{\mathrm{m}}{\mathrm{s}}$ in the outlet channel in which the nanofiber mat is placed.

To monitor mixing, the Species model is also enabled, which permits diffusion to be modeled on a mixture-type material, in which one species is dominant and the rest may be defined as possessing a certain mass fraction. Dilute species conditions, by which diffusion may be simulated via Fick's Law ($J = -D\frac{\delta c}{\delta x}$), are assumed in order simplify the simulation and prevent the material parameters of water from being affected by those of the specific species (such as water's viscosity being averaged with that of the species solution, or the solver using a multicomponent diffusion scheme in which water itself is also modeled as "diffusing" through the dissolved solute).

In the unit cube models, the sides of the cube parallel to the direction of flow (top, bottom, and the two sides) are assigned symmetry conditions in Fluent, which instruct the solver to remove the zero-slip wall conditions and treat them as frictionless surfaces, since they are not "walls," but simply boundaries beyond which more fluid lies in the actual nanofiber mat.

Lastly, in order to ensure that no programming language has been left out, selective modification of material-specific properties like diffusion coefficients in only specific regions of a model cannot be done using the default GUI in Fluent, and requires instead that a user-defined function (UDF) be written and compiled into the C programming environment on which Fluent is based. After quantifying the dispersion that occurs in the unit cubes, the following function was written with the purpose of replacing the *diffusion* coefficient of the species with the *dispersion* coefficient only within the nanofiber mat (here, $D = 1.0\times10^{-10}\frac{m^2}{s}$ and $E = 3.0 \times 10^{-9}\frac{m^2}{s}$; the nanofiber mat region is listed as possessing the zone_IDs 4 and 8 in Fluent):

```
#include "udf.h"
DEFINE_DIFFUSIVITY(diffusion_fun,c,t,i)
{ int zone_ID = THREAD_ID(t);

     real dd = 1e-10;
     if (zone_ID == 4 || zone_ID == 8){
          dd = 3e-8;
     }
     return dd;

}
```

The function was assigned in Fluent using the menu option: Define > User Defined... > Functions > Interpreted... after which a text file containing the function and saved with file extension .c was selected. It is somewhat understandable why material properties cannot, by default, be changed across in various geometries; since diffusion coefficients are material properties (as discussed during explanation of the dispersion model in equation 3 on page 55), they are defined by the material, and not by the zones/geometries through which the materials are flowing. However, in some special circumstances like these (and also for cases in which anisotropic magnetic/electric fields must be modeled), these modifications are necessary, and ANSYS's documentation of UDFs is very thorough.

5.6.2 Monitoring solution convergence

Convergence of a solution in Fluent was monitored simultaneously using the following three methods for each solution:

Monitoring Residuals: The scaled residuals, which are reported in the Scaled Residuals plot window when a solver begins iterating a solution, track the changes in the sum of the imbalances of each variable's conservation equations through each cell, and then scale this value over the entire model, by calculating and plotting the following equation [2]:

$$R_\phi = \frac{\sum_{cellsP} \left| \sum_{nb} a_{nb}\phi_{nb} + b - a_P\phi_P \right|}{\sum_{cellsP} |a_P\phi_P|} \tag{5}$$

where:

R_ϕ is the globally scaled residual for a variable,

a_{nb} is the normalized area of the faces shared with each neighboring cell (for which $\sum_{nb} a_{nb} = 1$), which ANSYS calls the "influence coefficient"

a_P is the center coefficient (likely dependent on cell size),

ϕ_{nb} and ϕ_P are the actual values for a general variable ϕ at cell P or at the cell neighboring cell sharing the face nb, respectively, and

b is the source term.

Thus, the conservation equation for a cell may be written as $\sum a_{nb}\phi_{nb} + b$, and the value of R_ϕ is dependent on the sums of the absolute differences between cell-center values and the values for which a cell's conservation equation with source terms do not account, all normalized by division by the term $\sum_{cells P} |a_P \phi_P|$. This gives meaning to the global residuals when assessing solution convergence. When a solution nears convergence, the values of cell variables become increasingly aligned with the values "expected" by the conservation equations defined by neighboring cells, and thus the rate at which the residuals decrease is slowed. A fair idea of convergence in the solution may be gained by watching the slope of these scaled residual plots; a leveling out of the residuals' slopes means the solution is reaching convergence, while a maintained steady decrease indicates that additional iterations are necessary to continue converging the solution. Residuals that *increase* consistently or show some other strange behavior indicate errors with the geometry, solver, or simply numerical errors if the residuals have already dropped sufficiently low.

Surface Monitors: In every solution, it is wise to create a monitor for a known quantity, like the outlet velocity of a fluid given a known the inlet velocity. A surface monitor is used to perform and then plot some calculation based on surface values, such mass-averaged velocity normal to the outlet. When the value of this quantity in the solution

stabilizes to the expected value, it is a sign that convergence–at least for the variable monitored by this plot–has been attained.

Manual contour observance: With some models, convergence may continue even after a sufficiently accurate extent of solution convergence has been reached, as the solution continues to iterate and further refine small differences in its cell variables. The contour plots of the surface monitors were saved every 10 iterations through macro-based automation in Fluent, and the results monitored as the solution progressed. The outputs of the manual contour convergence method for a 13-million-element fiber mesh are shown in Figure 15. Typically, for velocity/fluid-flow simulations, once the outlet-surface contour plots produced no noticeable/quantifiable difference between 4 or more updates (equivalent to 40 or more iterations), another 100 to 200 iterations were performed to ensure that no significant convergence was still occurring before the solution was deemed sufficiently accurate for iterations to be stopped, and the resulting solution deemed appropriate.

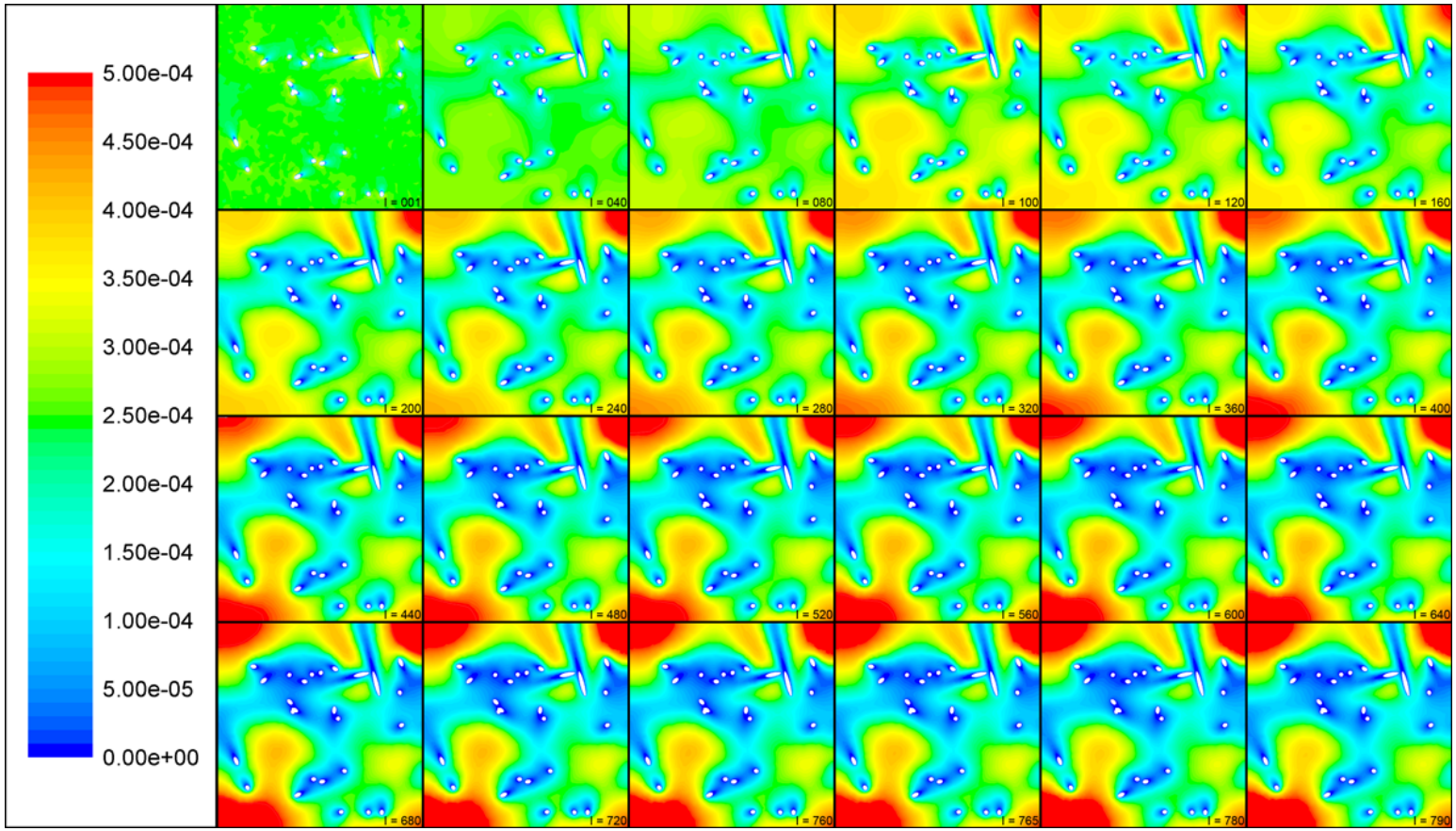

Figure 15: A sequence of images showing variation in the velocity magnitude contour plot in units of m/s at the outlet of a fiber unit-cube model as the solution converged. The process begins with almost no solution detail at the end of the first iteration, and slowly (over 790 iterations entirely with the Coupled solver) resolves the detail in the solution and reaches visible convergence. An additional 200 iterations were performed to confirm that the solution was sufficiently converged, while also monitoring the methods detailed in part 5.6.2 on page 66. Nanofibers exiting through the outlet plane produce the white circular outlines that are visible on the contour plot; as expected, low-velocity regions (shown in blue) develop around the proximity of the nanofibers, due to the non-slip boundary condition that was applied to all surfaces including the channel walls and the surface of each nanofiber (explained in part 5.2 on page 52).

6 Results

6.1 Flow in cube models

The following seven full-page figures (Figure 16 on the following page through Figure 22 on page 77), generated by exporting the results produced by Fluent into CFD-Post and rendering the results with streamlines and contour plots, show the behavior of the fluid flow in the fiber unit-cube models. These figures were all captured using the high-density, 13-million-element model in order to produce the smoothest streamlines for visualization purposes. As mentioned in the figure labels, the inlet faces were displayed, with the water inlet colored blue, and the solution inlet colored red (and containing 0.5 $\frac{units}{L}$ of a theoretical species with diffusivity $D = 1 \times 10^{-11} \frac{\mathrm{m}^2}{\mathrm{s}}$, as explained in part 5.3 on page 53). The color map utilized for all of the following images was the "FLUENT Rainbow" map from CFD-Post.

6.2 Quantifying Dispersion in Fiber Unit-Cube Models

Figure 23 on page 78 shows the input profiles for the diffusion cubes with diffusion coefficients of $6.0 \times 10^{-11} \frac{m^2}{s}$ to $50.0 \times 10^{-11} \frac{m^2}{s}$, with an interval of $0.5 \times 10^{-11} \frac{m^2}{s}$ between images. Figure 24 on page 79 and Figure 25 on page 80 illustrate sample results of the absolute-difference operation described in Figure 12 on page 59.

From the results of these operations, the data was processed as described in part 5.4 on page 59, and the plot in Figure 26 on page 81 was generated. The black line shows the average result computed for the fiber unit-cube models, and indicates that the most accurate dispersion coefficient to model the general characteristics of the fiber mat is $30 \times 10^{-11} \frac{m^2}{s}$, which is a 30-fold increase in effective diffusion relative to the baseline diffusion coefficient of $1.0 \times 10^{-11} \frac{m^2}{s}$, with which the fiber unit-cube models were simulated in the fiber unit-cube models.

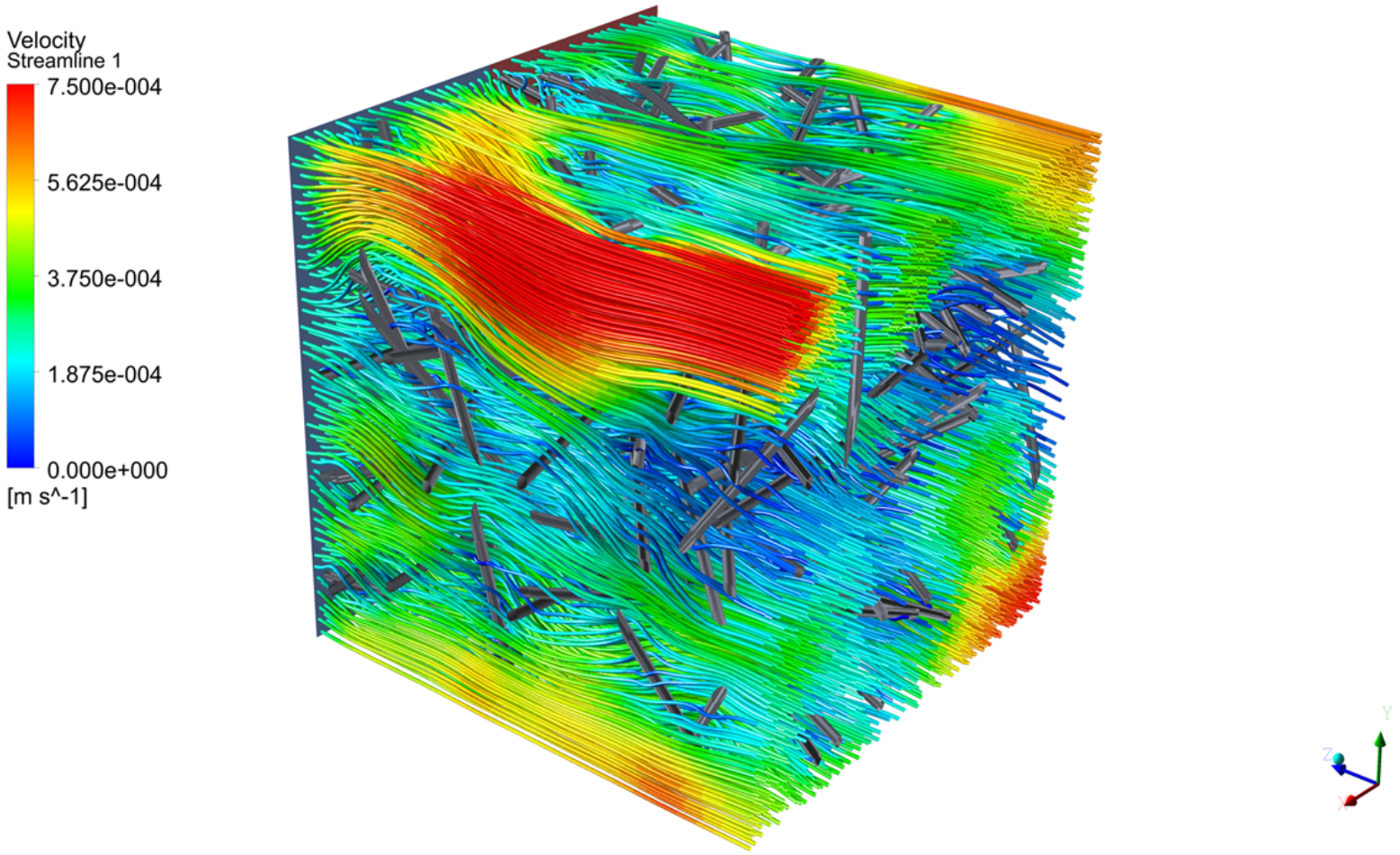

Figure 16: Flow streamlines (representing continuous laminar flow paths) in a fiber unit-cube model are displayed, and color mapped according to velocity magnitude, where red indicates relatively high velocity and blue indicates low velocity. Shown from a side perspective; the fluid is flowing away from the inlets shown on the left of the model, towards the right side of the figure, and exits the unit-cube model through the face directly opposite the inlets (the rightmost face on which the streamlines all end).

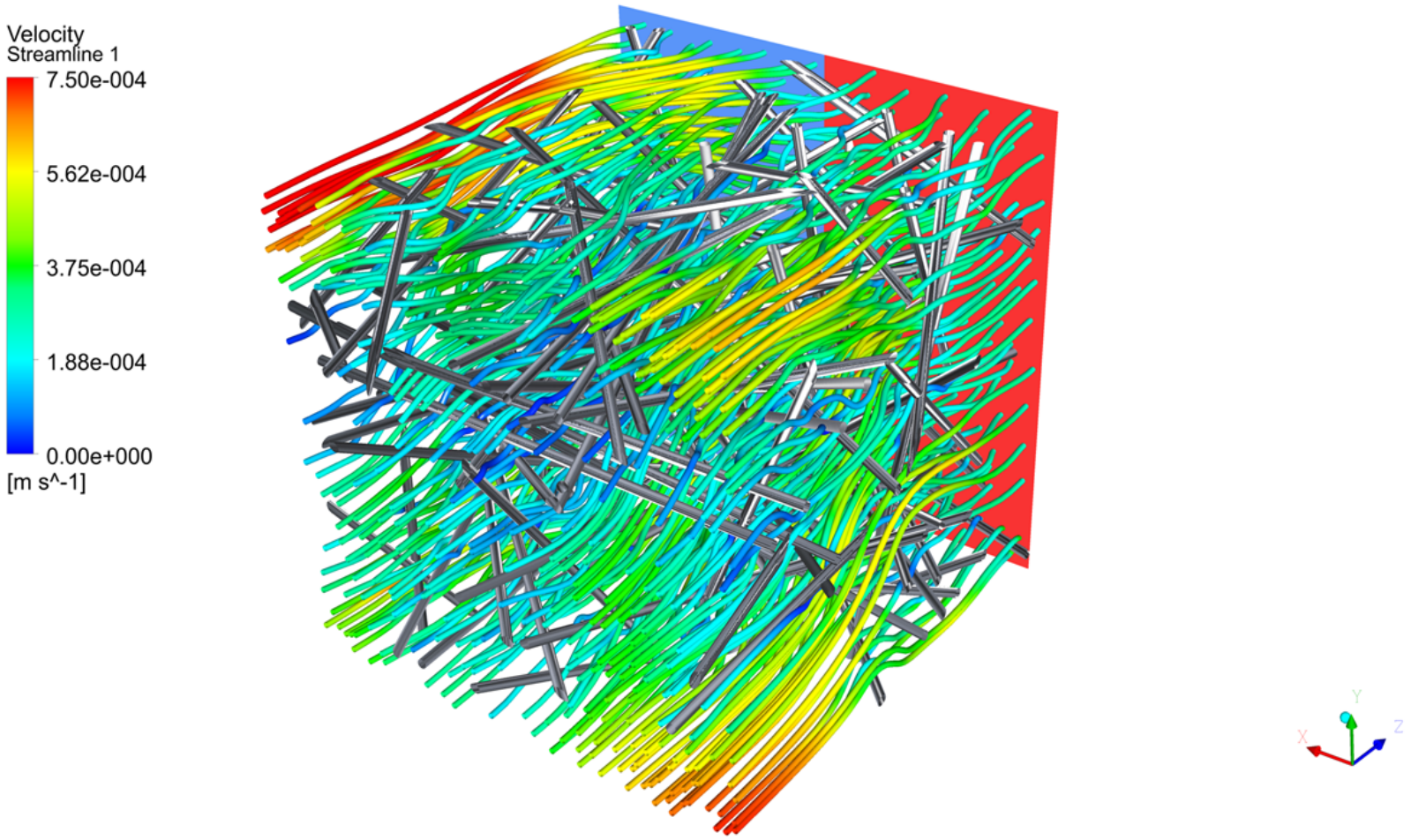

Figure 17: Flow streamlines (representing continuous laminar flow paths) are displayed, and color mapped according to velocity magnitude, where red indicates relatively high velocity and blue indicates low velocity, as shown in the color-map legend. The inlets are positioned at the back of the image, and fluid is flowing in the direction exactly opposite that in which the Z axis is pointing in the bottom right corner.

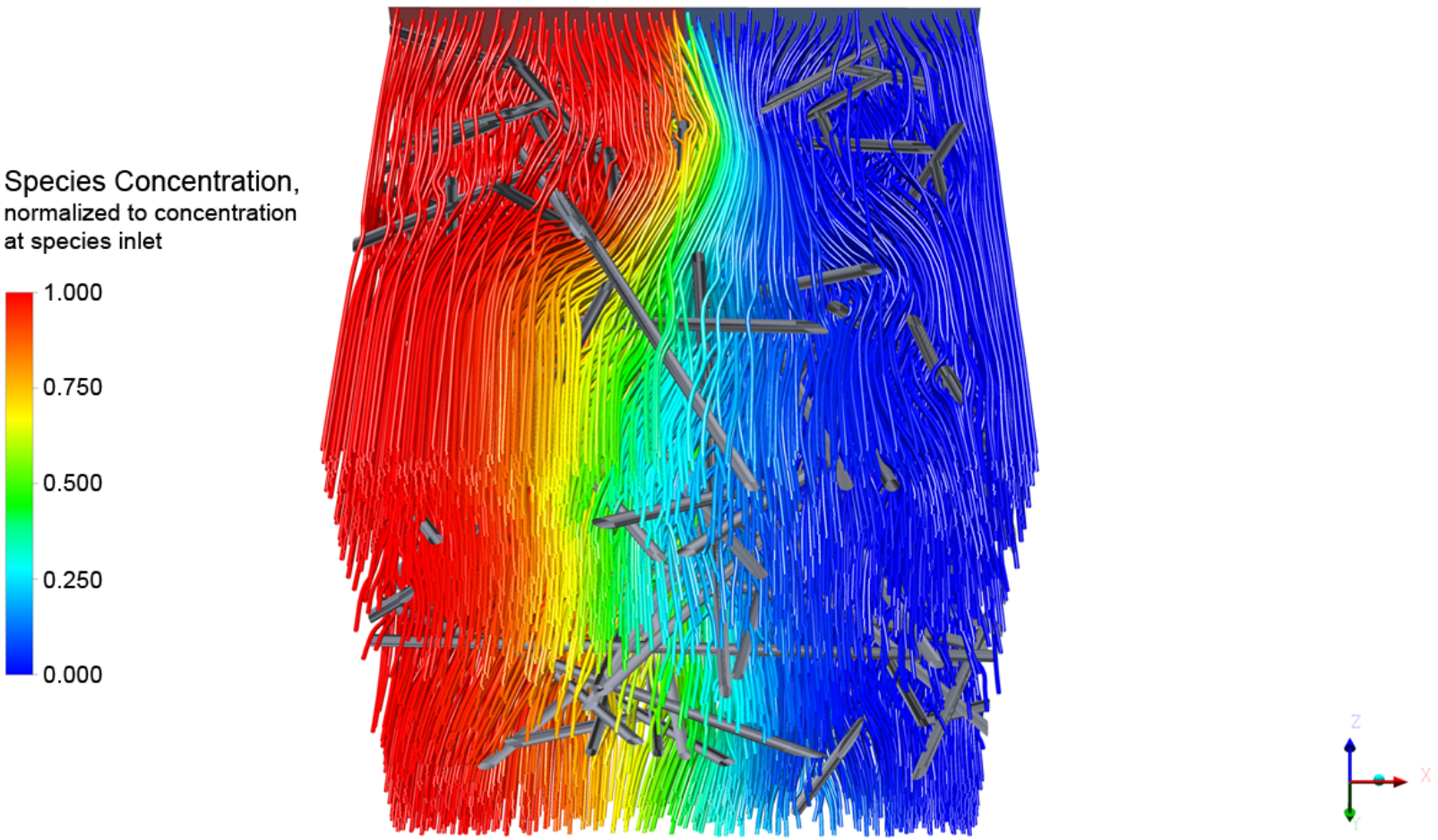

Figure 18: Flow streamlines colored by species transport, according to the displayed color map legend. The species concentration at the red (left) inlet is 0.5 $\frac{units}{L}$, and the concentration at the blue (right) face is 0 units/L. The color map in this figure and in the successive color figures is normalized to the species concentration at the species inlet (top left). Fluid is flowing away from the two-colored plane representing the inlet face of the cube, on which the leftmost face represents the dissolved-species inlet, and the rightmost face represents the water inlet.

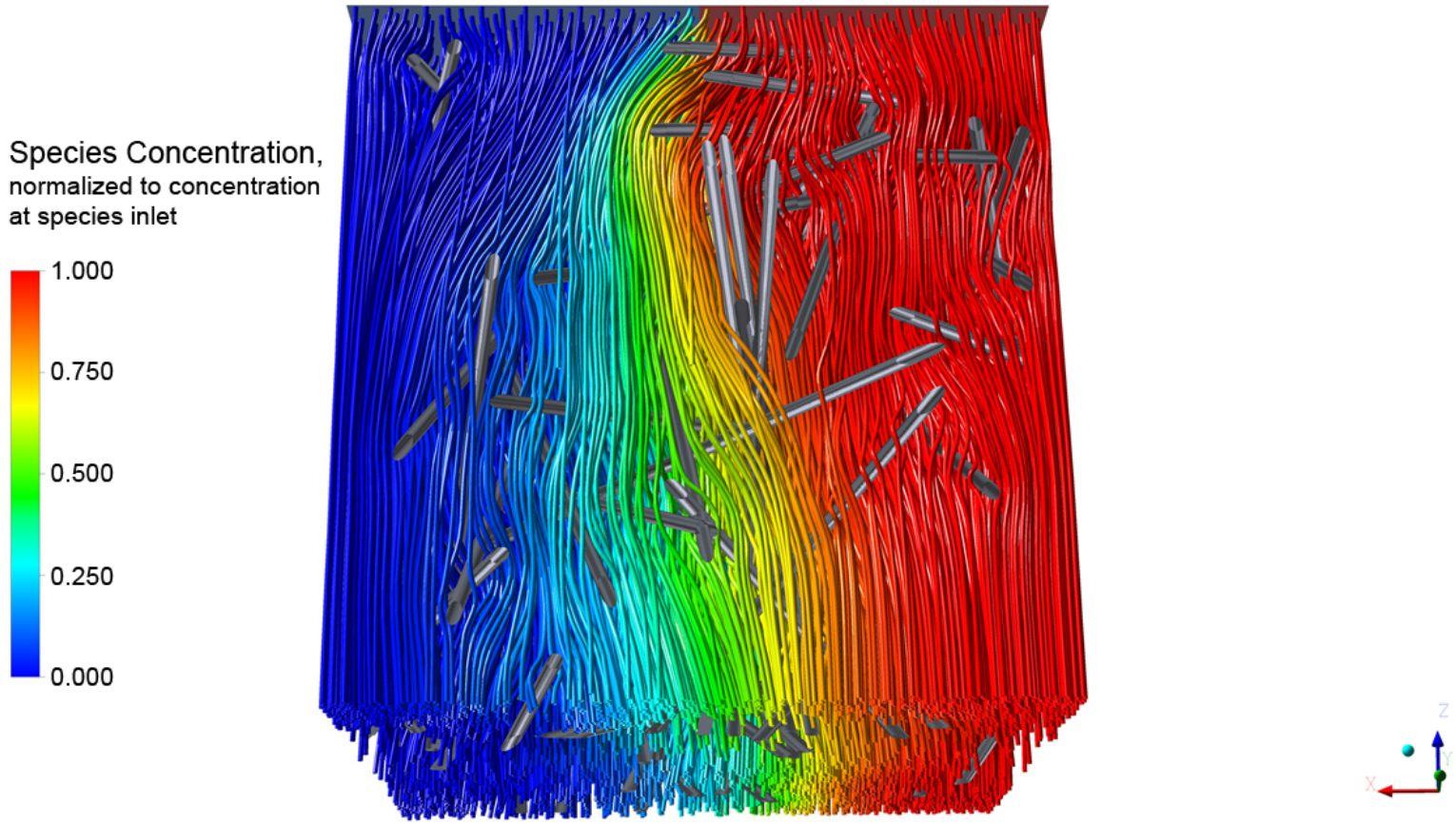

Figure 19: Flow streamlines colored by species transport, according to the displayed color map legend. The species concentration at the red (left) inlet is 0.5 units/L, and the concentration at the blue (right) face is 0 units/L. Fluid is flowing away from the visible inlet faces, with the dissolved-species inlet on the left colored red, and the water inlet on the right colored blue. Similar to Figure 18 on the preceding page, except rotated to display a bottom-up view that to better highlights the dispersion profile.

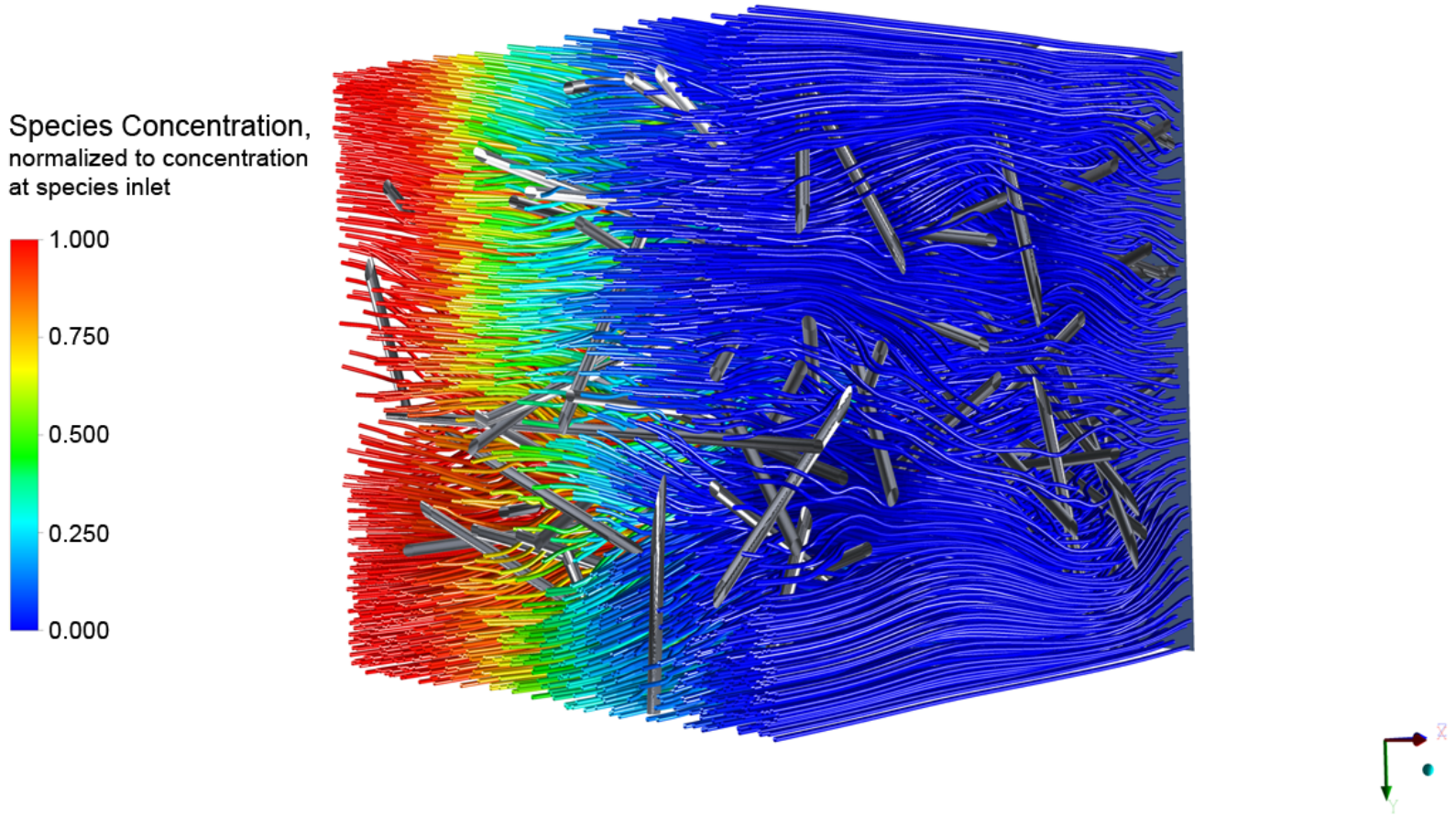

Figure 20: Flow streamlines colored by species transport, according to the displayed color map legend. Provides a view of the flow paths in the side profile, and also shows the variation in streamline paths along the dispersion profile.

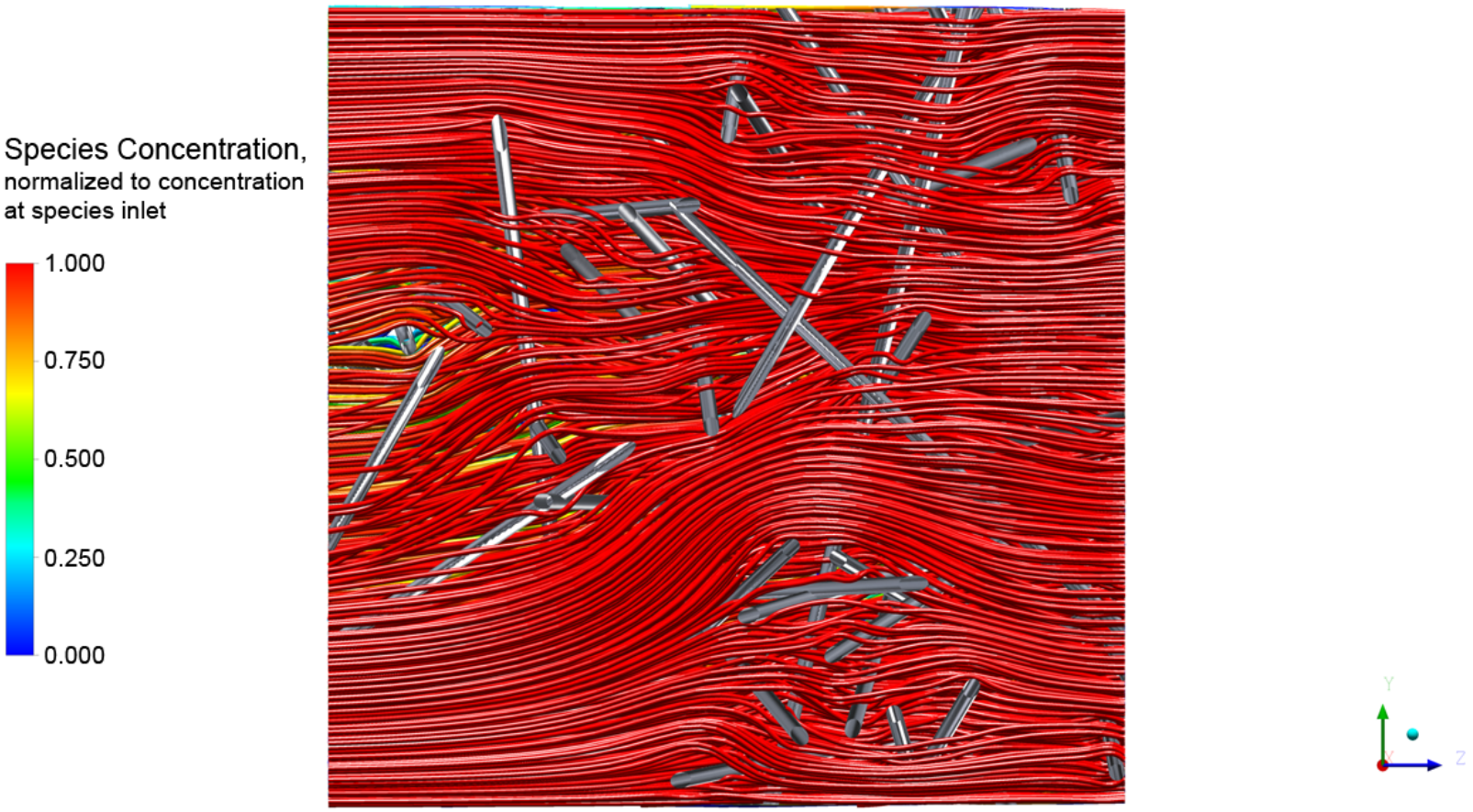

Figure 21: Flow streamlines colored by species transport, according to the displayed color map legend, illustrating the varied streamlines in the side profile of the species solution, with fluid flow traveling from the right side of the model to the left (opposite to the "Z" direction overlaid in blue on the axis in the bottom right corner of the image).

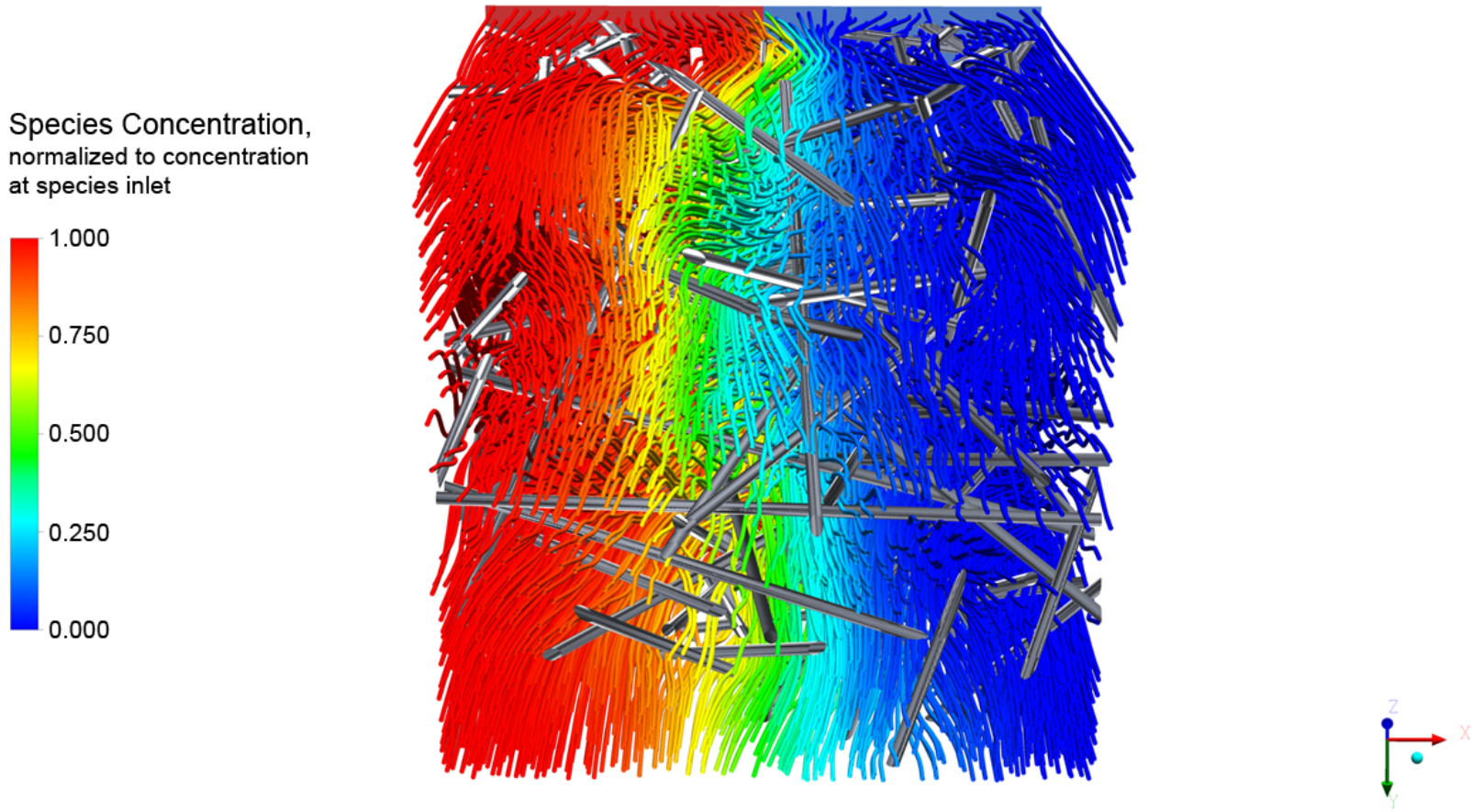

Figure 22: Flow streamlines colored by species transport, according to the displayed color map legend. The fluid flow is traveling from the red species-inlet and blue water-inlet planes, outwards from the page. This streamline once again captures the dispersive effects caused by percolation of water around the fibrous channels.

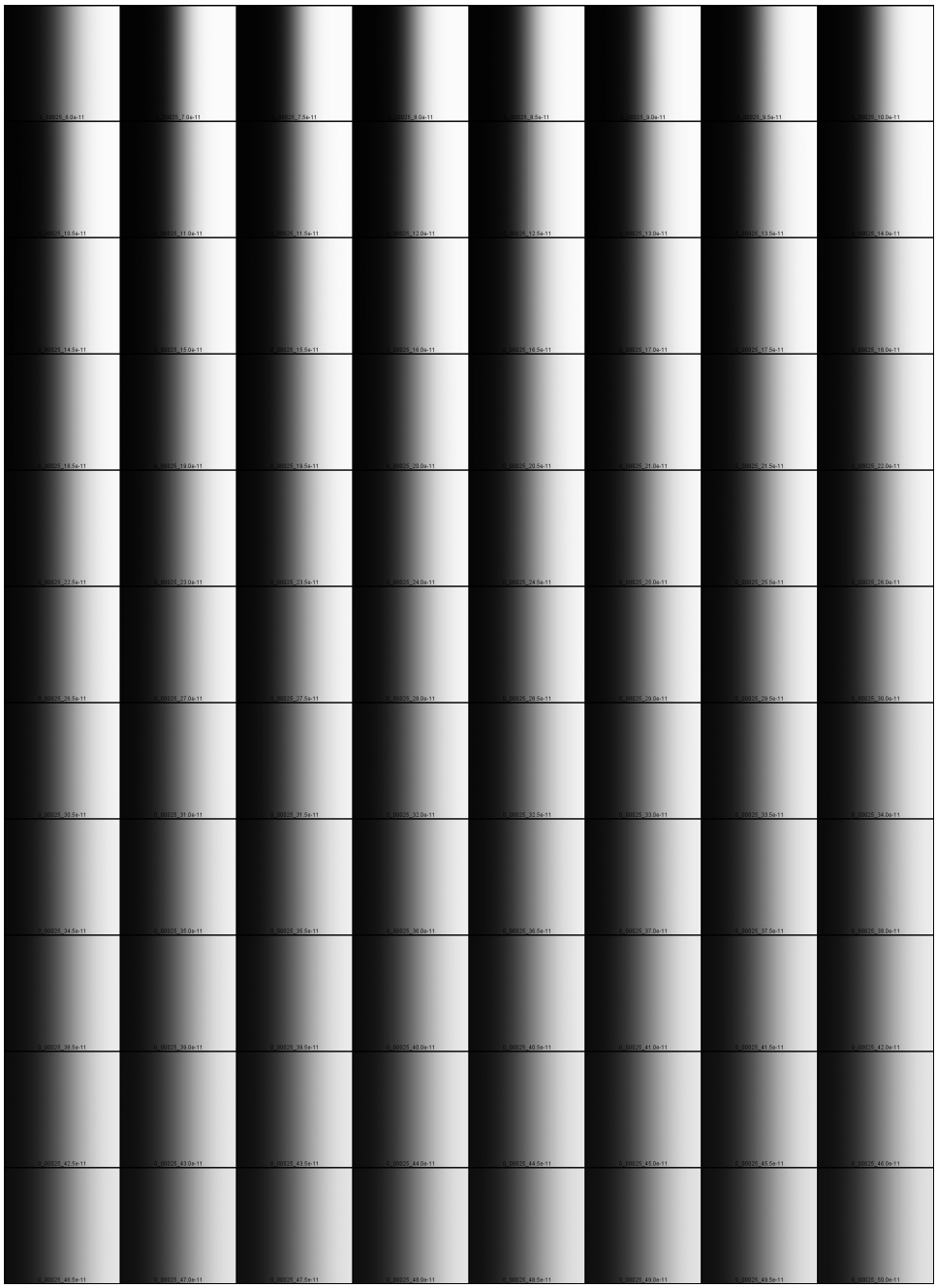

Figure 23: Sample results from the diffusion cubes, in which a grayscale intensity map is applied to quantify the different species-concentration profiles which result from each of the diffusion coefficients (D). In this series of images, a generalized species concentration of $0.5\frac{\text{units}}{\text{L}}$ was flowed from one inlet in the unit cube and water ($0.0\frac{\text{units}}{\text{L}}$ of solution) was flowed through the other inlet. White represents a concentration of $0.5\frac{\text{units}}{\text{L}}$, and black a concentration of $0.5\frac{\text{units}}{\text{L}}$. The ranges of D shown are $D = 6.0 \times 10^{-11}\frac{m^2}{\text{s}}$ through $D = 49.0 \times 10^{-11}\frac{m^2}{\text{s}}$, with intervals of $0.5 \times 10^{-11}\frac{m^2}{\text{s}}$ separating each individual image above.

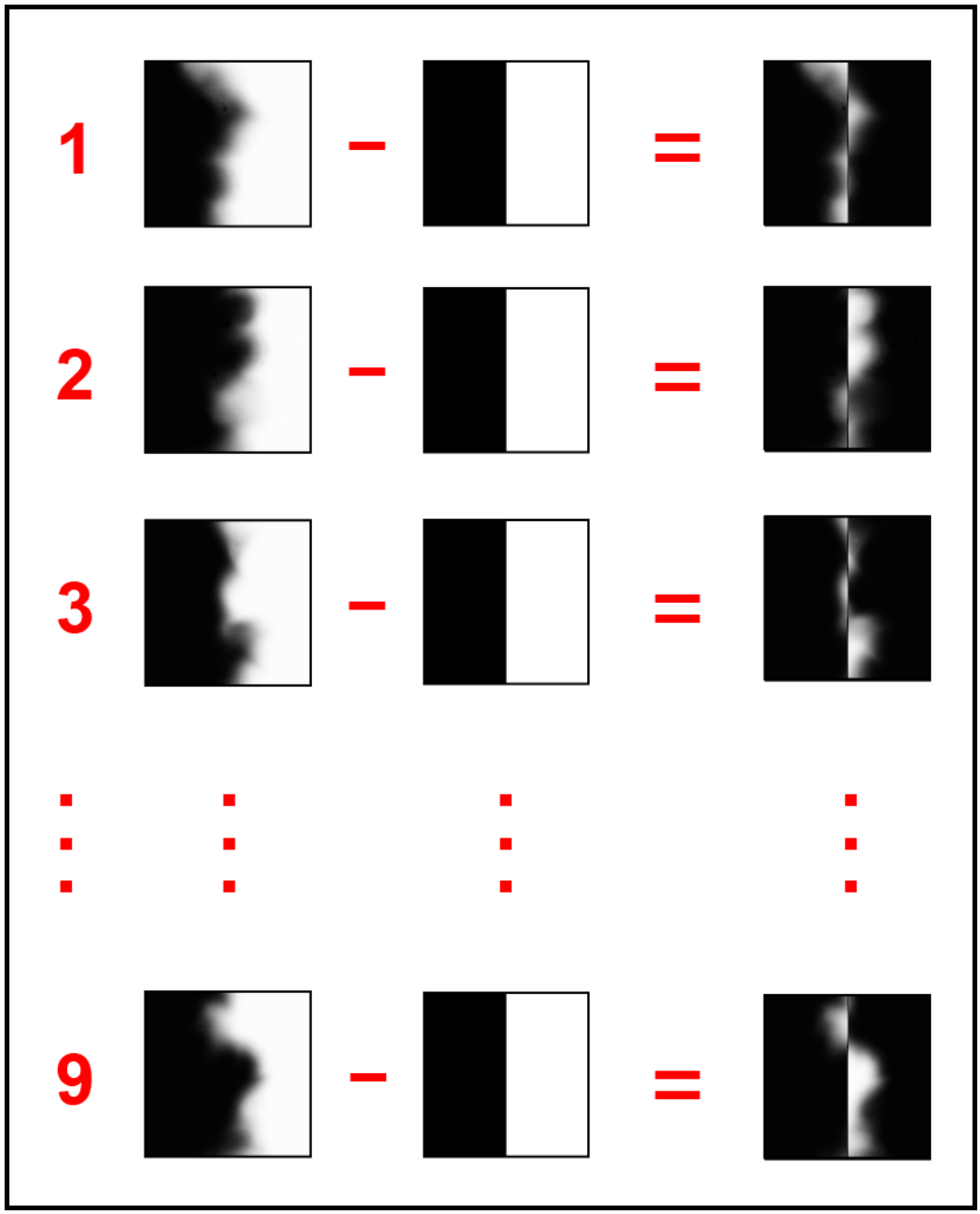

Figure 24: The process by which the absolute-difference array of mass-transport data for flow in each of the 9 nanofiber unit-cubes was calculated is shown above, according to the method specified in part 5.3.2. The numbers in the leftmost column identify the nine distinct fiber unit-cube models used to generate dispersion profiles. The differences in the concentration contour plots at the outlets of the unit-cube models are a result of the different fiber geometries that each model contained.

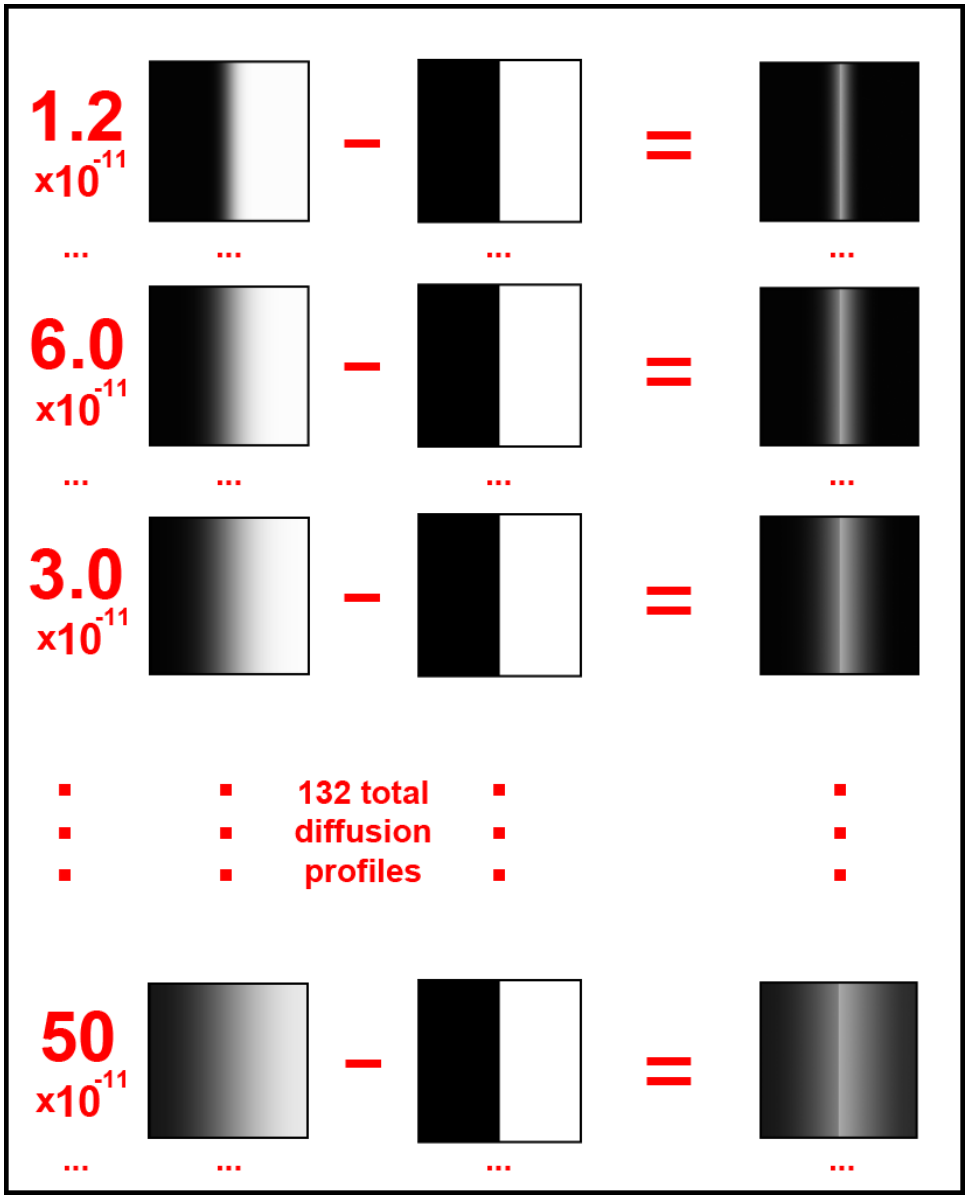

Figure 25: The process by which the absolute-difference profiles for diffusion-only species transport is shown, according to the method specified in part 5.3.2 on page 56. The resulting data in the third column represents the extent to which of lateral species transport was generated by the different diffusion coefficients. The leftmost column of numbers indicates the species diffusion coefficient used in each model. The summed intensity of all pixels in each of the absolute-difference images in the rightmost column represents the total species transport produced by the various diffusion coefficients; the purpose of these fiberless models is to identify what value for the diffusion coefficient produces an equal amount of *diffusive* species transport as is produced through *dispersive* species transport in the fiber unit-cube models, since this is the value that is necessary to model dispersion using equation 3 on page 55.

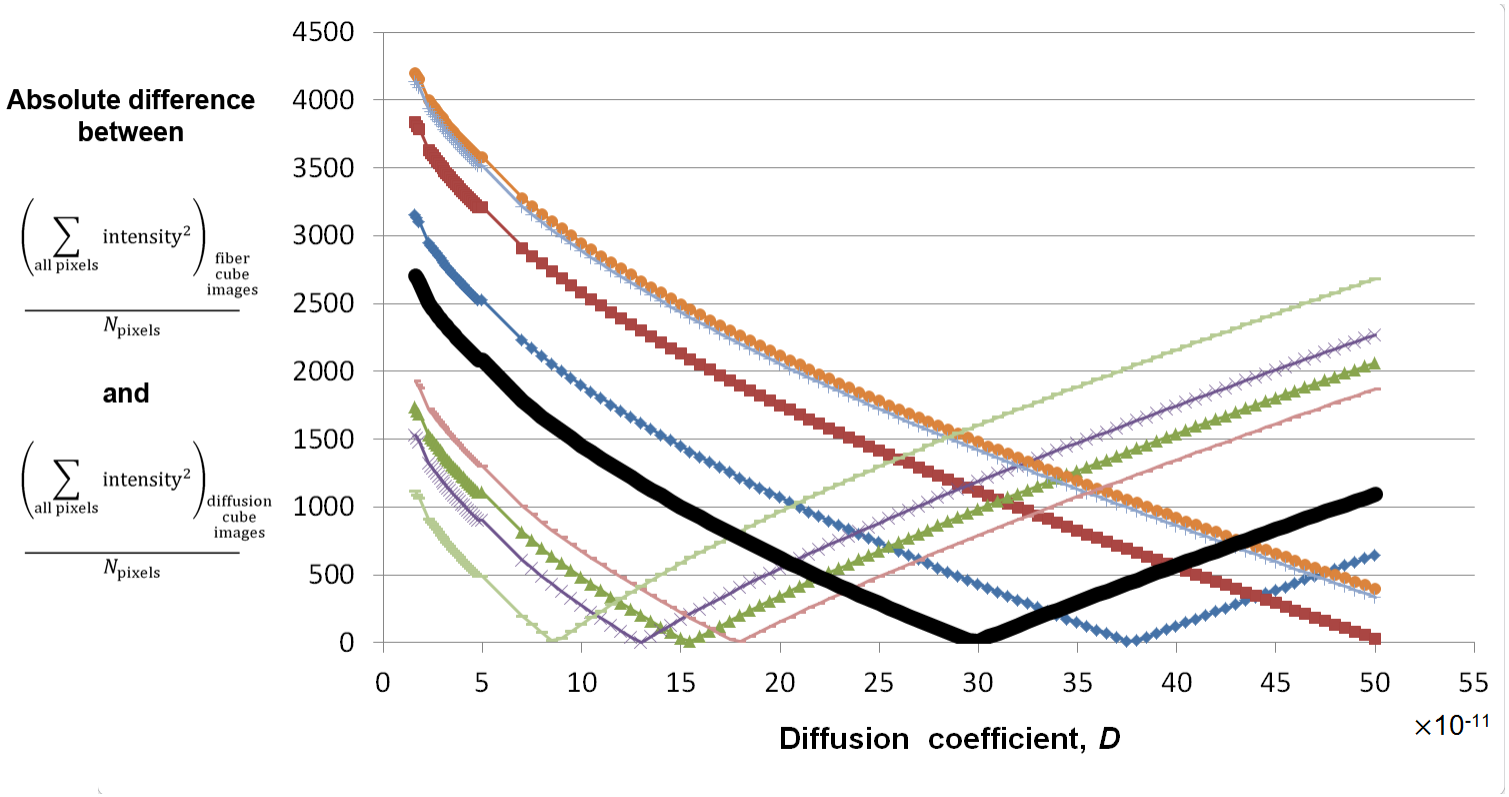

Figure 26: The absolute differences between the sum of squared pixel intensities in the processed fiberless diffusion cubes and the processed fiber unit-cube models are plotted. "Processed" refers to the steps outlined in Figure 5.3.3 on page 57 and Figure 5.3.2 on page 56. Minimum values represent the dispersion coefficient that most accurately describes the flow in each fiber system. Results of each fiber unit-cube models are plotted in color, and the average value, whose minimum corresponds to $D = 30 \times 10^{-11} \frac{m^2}{s}$, is represented by a thick black line. Note that these best-fit values for the dispersion coefficient are relative to a baseline diffusion coefficient of $D = 1 \times 10^{-11} \frac{m^2}{s}$ in the fiber unit-cube models; this means that the best-fit result $D = 30 \times 10^{-11} \frac{m^2}{s}$ indicates that dispersion enhances mixing by a factor of 30, and not that the dispersion is fixed to this precise value. That is, the effective dispersion coefficient, E, may simply be defined as $E = 30D$ for values of D within reason.

6.3 Mixing Simulations in the Y-Shaped Microfluidic Channel

Using the 30-fold factor that represents the effective increase in dispersion within the nanofiber zone relative to a dissolved species' diffusion coefficient in the open channels up- and downstream of the fiber mats, the following images were generated: Figure 27 shows a contour profile of velocity on the XZ-axis symmetry plane at the center of the channel; Figures 28 and 29 show the concentration profile that occurs in empty channel with no nanofiber mats, for diffusion coefficients of $1 \times 10^{-11} \frac{m^2}{s}$ and $3 \times 10^{-10} \frac{m^2}{s}$, respectively; Figures 30 and 31 show the concentration profile of the channel with a generic species that possesses a diffusion coefficient of $1 \times 10^{-9} \frac{m^2}{s}$, which increases to $3 \times 10^{-8} \frac{m^2}{s}$ within the nanofiber-mat portion of the channel.

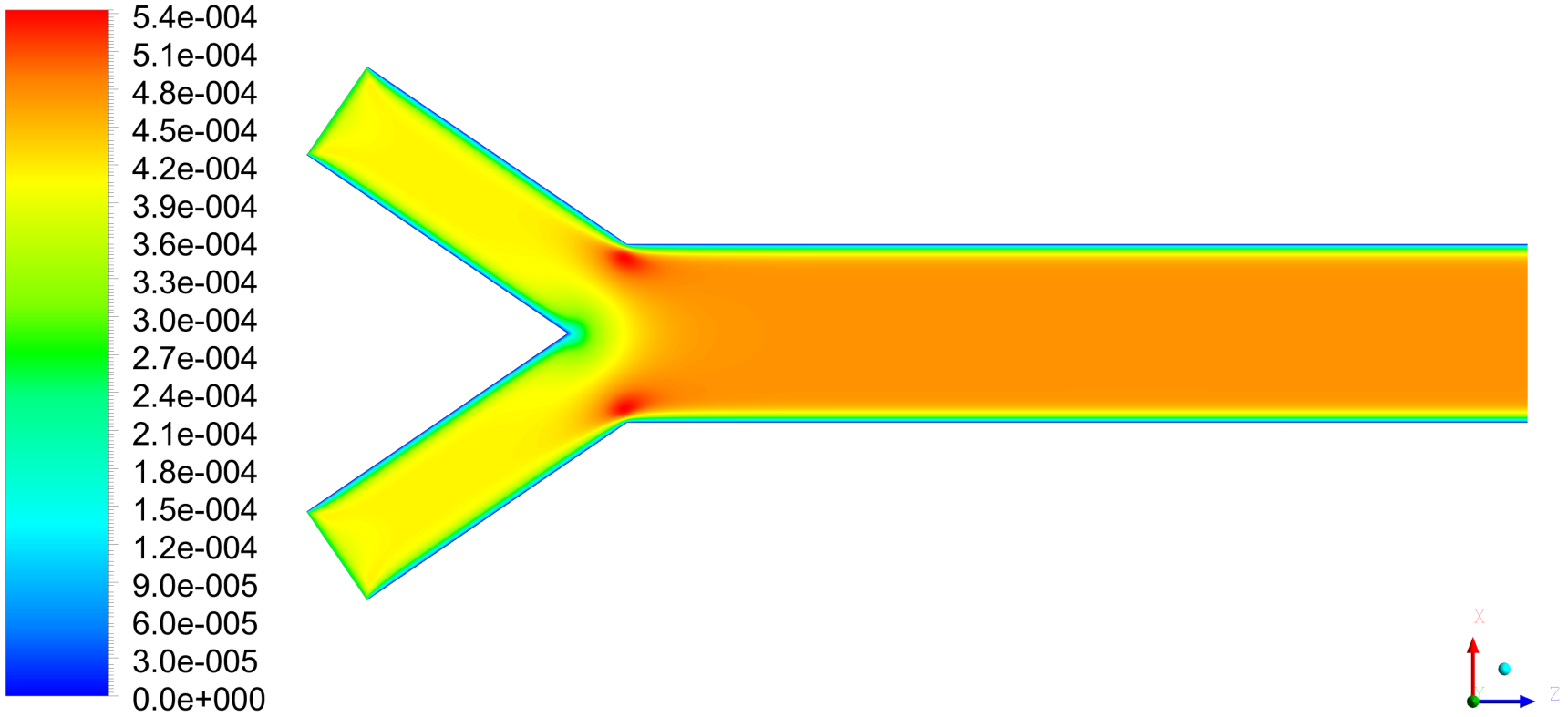

Figure 27: Top view of the center plane of the Y-channel, showing the velocity profile at the center of the channel. No-slip boundary conditions were applied to the walls of the channel, and this is shown by the thin blue contour that spans all edges of the channel. Units displayed in $\frac{m}{s}$.

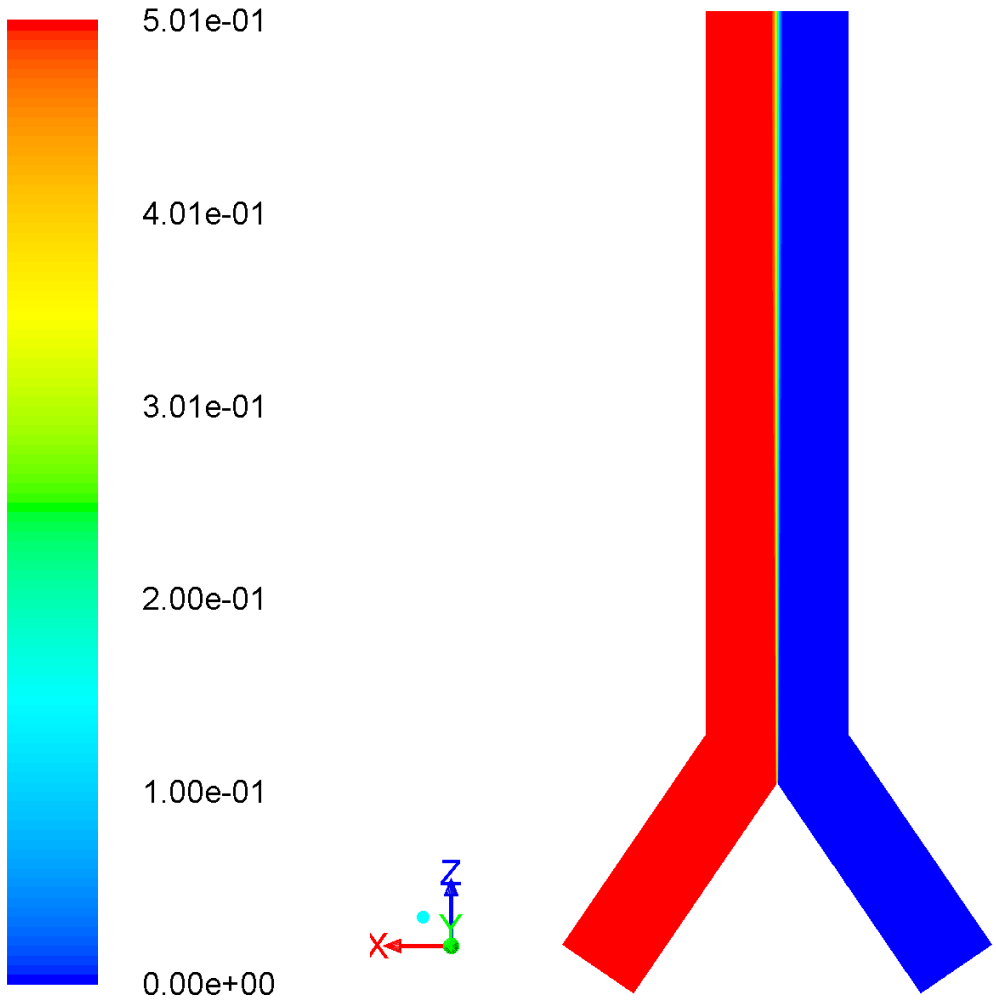

Figure 28: Top view of the center plane of the Y-channel, in which no nanofiber was placed. The total outlet flow is 1.5 $\frac{\mu L}{min}$, and the dissolved species was given a generic concentration of 0.5 $\frac{units}{L}$ at the species inlet (shown in red) and 0 $\frac{units}{L}$ at the at the pure-water inlet (shown in blue). The diffusion coefficient of the species was set to $1 \times 10^{-11} \frac{m^2}{s}$ throughout the channel. This figure illustrates the extremely low mixing rate for biological specimens, for which the diffusion coefficients are in the 10^{-11} order of magnitude.

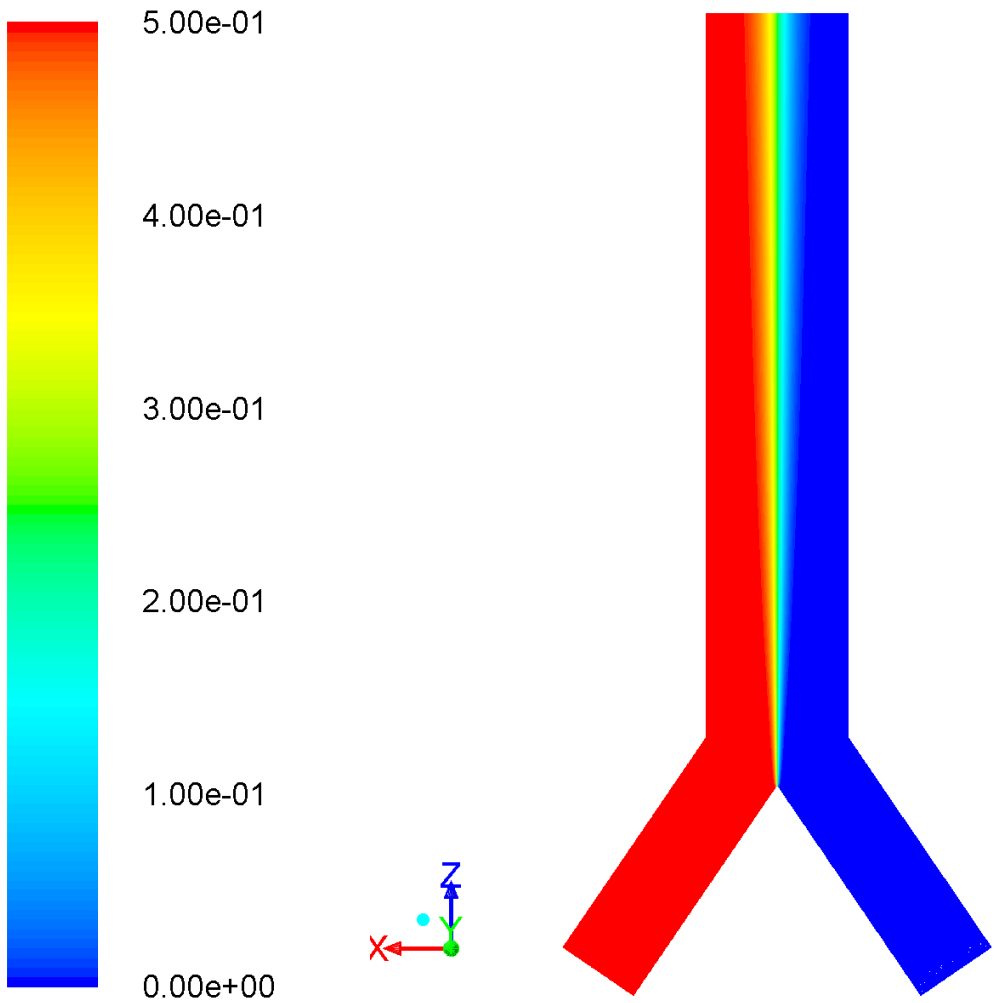

Figure 29: Top view of the center plane of the Y-channel, in which no nanofibers were placed. The total outlet flow is 1.5 $\frac{\mu L}{min}$, and the dissolved species was given a generic concentration of 0.5 $\frac{units}{L}$ at the species inlet (on the left side, shown in red) and 0 $\frac{units}{L}$ at the at the pure-water inlet (on the right side, shown in blue). The diffusion coefficient of the species is set to $3 \times 10^{-10} \frac{m^2}{s}$ throughout the channel. This figure illustrates the relatively low extent of mixing in an empty channel.

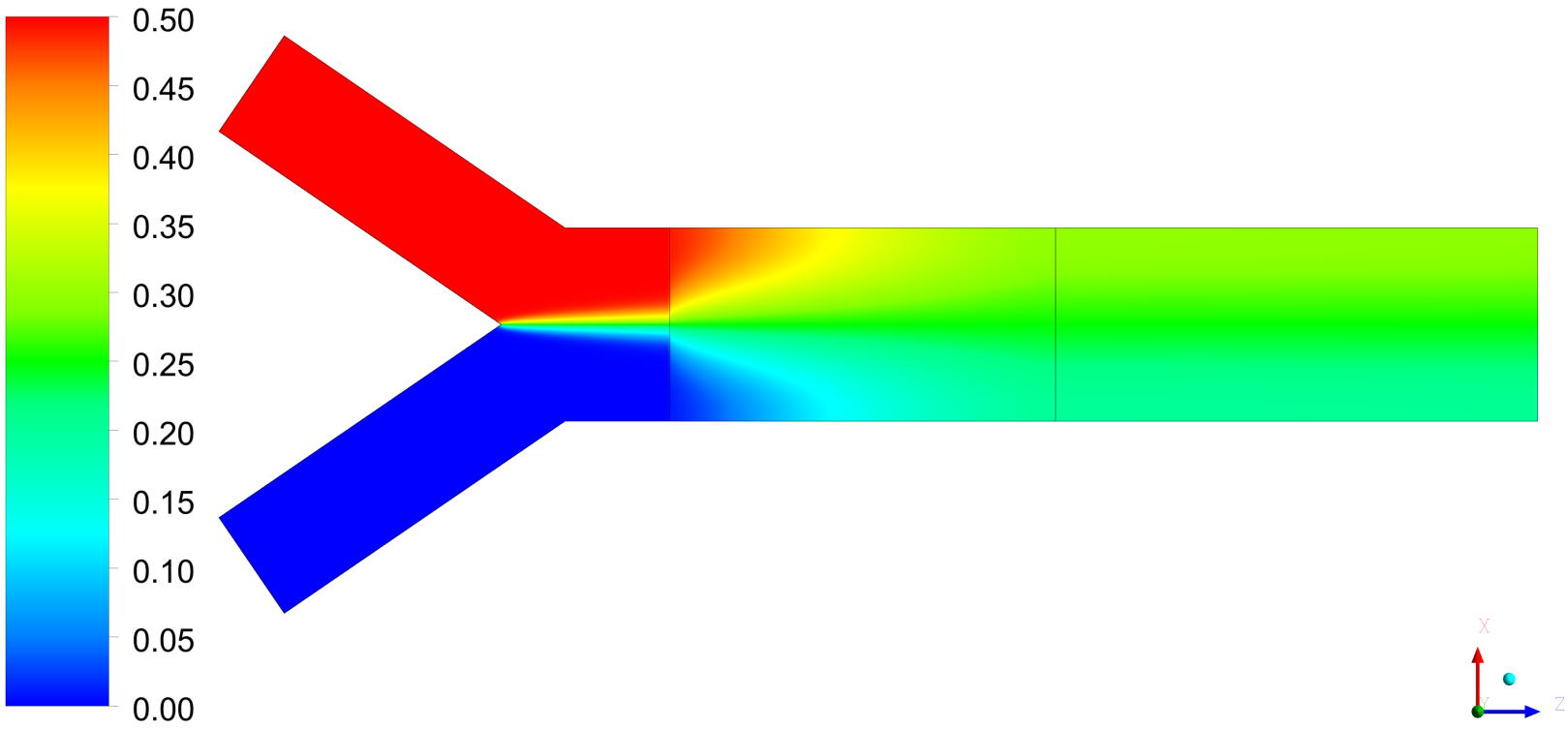

Figure 30: Top view of the center plane of the Y-channel, showing the rapid mixing that occurs at the boundary of the nanofiber mat. Total outlet flow is $1.5\frac{\mu L}{min}$, and the dissolved species is given a generic concentration of $\frac{0.5\,units}{L}$ at the species inlet (shown in red) and $\frac{0\,units}{L}$ at the at the pure-water inlet (shown in blue). The diffusion coefficient of the species is set to $1 \times 10^{-9}\frac{m^2}{s}$ outside of the fiber mat, and is increased 30-fold to $3 \times 10^{-8}\frac{m^2}{s}$within the microchannel. Black outlines mark the start and end of the nanofiber mat.

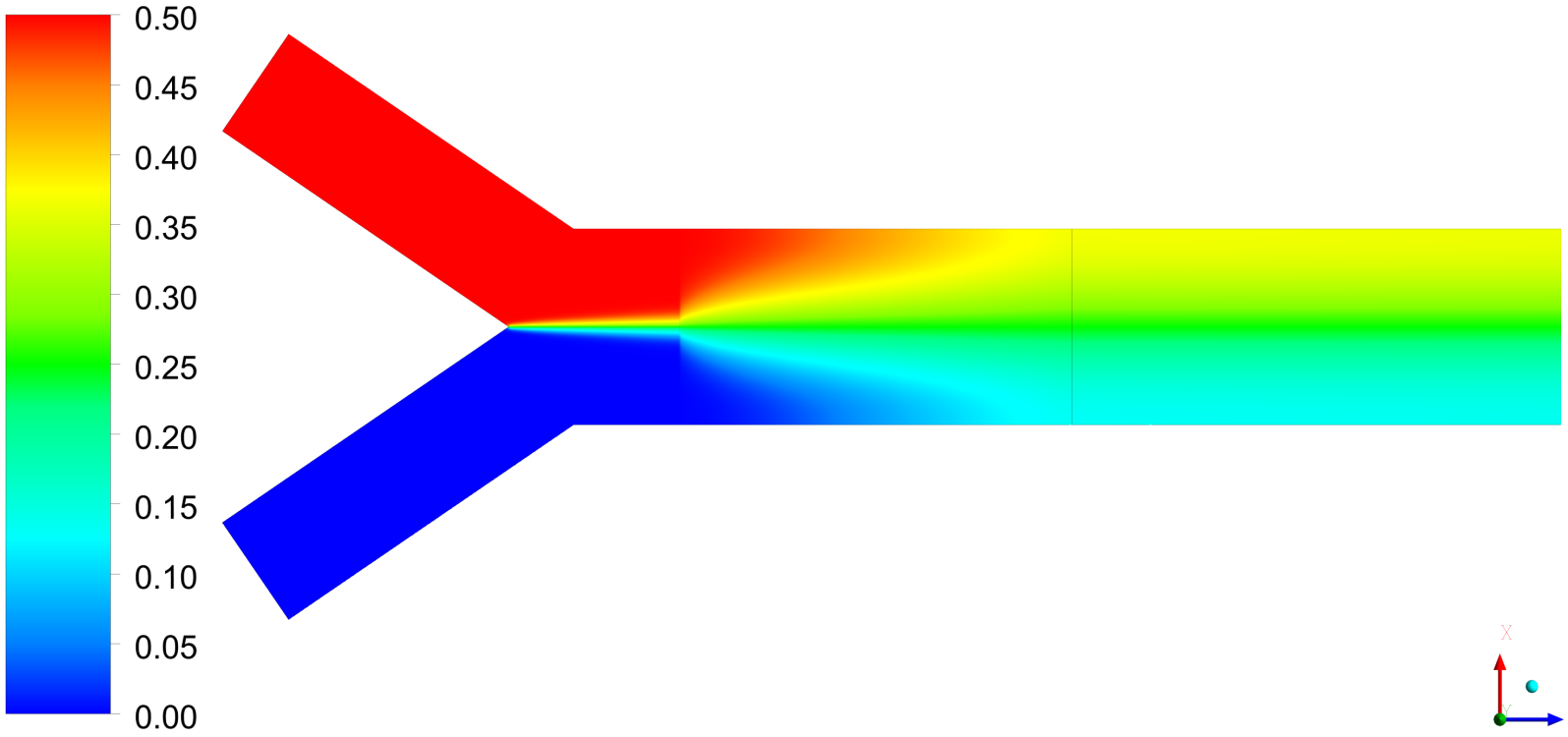

Figure 31: Top view of the center plane of the Y-channel, similar to Figure 30 on the preceding page except with the flow rate doubled to $3.0\frac{\mu L}{min}$. The dissolved species is given a generic concentration of $\frac{0.5\,units}{L}$ at the species inlet (shown in red) and $\frac{0\,units}{L}$ at the pure-water inlet (shown in blue). The diffusion coefficient of the species is set to $1 \times 10^{-9}\frac{m^2}{s}$ outside of the fiber mat, and is increased 30-fold to $3 \times 10^{-8}\frac{m^2}{s}$within the microchannel. Black outlines mark the start and end of the nanofiber mat.

7 Discussion

7.1 Analysis of Results

To summarize, the Y-channel mixing problem defined in part 5.1 on page 52 was modeled using the ANSYS Fluent solver, by first simulating the fluid flow through several small cubic samples that were assumed to be representative of the overall nanofiber structure, and then applying the average dispersion coefficient that was elucidated from these fibers to the nanofiber mat. A user-defined function in Fluent was necessary to assign the dispersion coefficient to the fiber mat zone, and after the case setup was performed, the solution was iterated to conversion. In this report, the entire design and simulation process of the Y-channel was presented as an example case on which the workflow for modeling the electrospun nanofibers in ANSYS could be demonstrated.

As is visible from a comparison between the results of Figure 30 on page 86, in which the nanofiber mat is simulated, and Figures 28 on page 84 and 29 on page 85 in which no nanofiber mats are available to enhance mixing, the dispersion-enhanced mixing generated by the nanofibers causes a very dramatic effect, and induces significant amounts of mixing in only a 2-mm long section of fibers. In reality, the nanofiber mat is spun to a length of at least 5-10 mm in the channel, thus ensuring that the mixing performance from these simulations is, if anything, an under-representation of the real-world mixing results that would be produced by a nanofiber mat of appropriate length.

7.2 Future Research

Automating the majority of this process, with the only exception being meshing (which is simply performed by right-clicking on the Mesh cell in Workbench and choosing Update before continuing with Fluent automation), facilitates many otherwise tedious tasks. For example, the effects produced by various modifications to the Y-channel design, such as varying the channel thickness or modifying inlet placement, may be quickly assessed. In addition, the

automation could be used for an optimization workflow, in which ideal geometries or physical parameters can be elucidated by comparing a wide range of test models and analyzing their associated results. For example, the mixing efficiency of baffle-type passive mixing elements could be assessed through several incremental changes to their layout, and the ideal placement that produces the most mixing could be easily identified.

This research will also be continued through a more thorough analysis of the dispersion generated by the unit cubes. The effects of increasing and decreasing the size of the unit cube will be more closely examined, along with the effects of fiber density and the introduction of fiber curvature. Much time was spent programming implementations of the algorithms that generate realistic curved nanofibers (shown in Figure 5 on page 34, for example), and the decision to simplify the dispersion models by using only linear fibers should be be reconsidered. Moreover, the orientation of the fibers within the unit cube should be altered to more accurately reflect the more planar and stacked orientation of the fibers in the actual fiber mat.

Quantifying the extent to which mixing is improved by the fibers is also essential; currently, the presented results are only qualitative, presented visually in the top-down contour plots of the Y-channel. It is clear that mixing is occurring in these figures, as illustrated by a comparison of the narrow diffusion profiles observed in empty channels with the far more homogenous species-concentration profiles produced by flow through the channels *with* nanofiber mats. To quantify and better define this mixing behavior, extent-of-mixing calculations will be performed on cross-section contour plots of the species concentration in the channel.

After the computational models are refined as discussed above, it will ultimately be necessary to compare these results with experimental data. If the simulated mixing performance diverges from experimental results, the experimental results may be used to troubleshoot any errors in the simulation. On the other hand, if the simulation results *mirror* experimental data, conditions should be varied until the simulated model becomes inaccurate, in order to

determine its limitations.

8 Appendix

On the next several pages, the scripts that were utilized specifically in the case file are presented, with some programs truncated to conserve space. If the portions of the following functions are used by a reader of this document, it is important that their syntax be double-checked to avoid any errors, and that all variables used within these scripts be declared, if their declaration is not already made in the snippets below.

8.1 Snippets of the Fiber Coordinate Generation Script - Java

```
private static double[] generatePointOnFace(int faceID) {
     double x1, y1, z1;
     if (faceID == 1) {
          x1 = x_origin + (x_length - x_origin) * rand.nextDouble();
          y1 = y_origin + (y_length - y_origin) * rand.nextDouble();
          z1 = z_origin - bufferDistance;
     }  else if (faceID == 2) {
// generate point on XZ plane (y = 0)
x1 = x_origin + (x_length - x_origin) * rand.nextDouble();
y1 = y_origin - bufferDistance;
z1 = z_origin + (z_length - z_origin) * rand.nextDouble();
}  else if (faceID == 3) {
// generate point on YZ plane (x = 0)
x1 = x_origin - bufferDistance;
y1 = y_origin + (y_length - y_origin) * rand.nextDouble();
z1 = z_origin + (z_length - z_origin) * rand.nextDouble();
}  else if (faceID == 4) {
// generate point on XY-far plane (z = z_length)
x1 = x_origin + (x_length - x_origin) * rand.nextDouble();
y1 = y_origin + (y_length - y_origin) * rand.nextDouble();
z1 = z_length + bufferDistance;
}  else if (faceID == 5) {
// generate point on XZ-far plane (y = y_length)
x1 = x_origin + (x_length - x_origin) * rand.nextDouble();
y1 = y_length + bufferDistance;
z1 = z_origin + (z_length - z_origin) * rand.nextDouble();
}  else { // faceID = 6
// generate point on YZ-far plane (x = x_length)
x1 = x_length + bufferDistance;
y1 = y_origin + (y_length - y_origin) * rand.nextDouble();
z1 = z_origin + (z_length - z_origin) * rand.nextDouble();
}
        double[] faceCoords = {x1, y1, z1};
return faceCoords;
} // end of generatePointOnFace(int faceID)
public static void generateStraightFibers() {
double[] startFaceCoords = new double[3];
double[] endFaceCoords = new double[3];
for (int curveID = startCurveIndex; curveID < numberOfFibers + startCurveIndex; curveID++){
        int startFace = rand.nextInt(6) + 1;
int endFace = rand.nextInt(6) + 1;
while (endFace == startFace) {
            endFace = rand.nextInt(6) + 1;
    }
    startFaceCoords = generatePointOnFace(startFace);
    endFaceCoords = generatePointOnFace(endFace);
    fibers[curveID].setPointCoords(1, startFaceCoords[0], startFaceCoords[1], startFaceCoords[2]);
    fibers[curveID].setPointCoords(2, endFaceCoords[0], endFaceCoords[1], endFaceCoords[2]);
}
}
public static void generateCurvedFibersOrigin() {
    for (int curveID = startCurveIndex; curveID <= numberOfFibers + startCurveIndex; curveID++) {
        // curve loop -- defines the Curves within the fibers[] array
        fibers[curveID] = new Curve(curveID, pointsPerFiber);
        fibers[curveID].setPointCoords(1, x_origin, y_origin, z_origin);
        for (int pointID = 2; pointID <= pointsPerFiber; pointID++) {
            coordinates = generateCoordsOnSphere(sphereRadius);
            fibers[curveID].setNextCoords(pointID, coordinates[0], coordinates[1], coordinates[2]);
        }
    }
}
public static double[] generateCoordsOnSphere(double radius) {
    double x, y, z;
    double radSquared = radius * radius;
```

```
    x = Math.random()*radSquared;

    y = Math.random()*(radSquared - x);

    z = radSquared - x - y;


    x = Math.sqrt(x);
    y = Math.sqrt(y);
    z = Math.sqrt(z);

    if (Math.random() >= 0.5) { // 50% chance that x is negative
     x = -x;
    }
    if (Math.random() >= 0.5) { // 50% chance that y is negative
     y = -y;
    }
    if (Math.random() >= 0.5) { // 50% chance that z is negative
     z = -z;
    }
    double[] coords = {x, y, z};
    return coords; // returns the coordinates in a double array,
                   // in which coords[0] stores x, coords[1] stores y, and coords[2] stores z.
} // end of generateCoordsOnSphere(double radius)
public static void printResults() {
    PrintWriter result;
    try {
        result = new PrintWriter(new FileWriter("C:/Users/.../FiberData.txt"));
    }
    catch (IOException e) {
        System.out.println("Can't save file result.txt!");
        System.out.println("Error: " + e);
        return; // End the program.
    }
    for (int i = 1; i <= numberOfFibers; i++) { // curve loop
        int numPoints = fibers[i].getNumPoints();
        for (int j = 1; j <= numPoints; j++) {
            result.println(i + " " + j + " " + fibers[i].getStringCoords(j));
        }
        result.println();
    }
    result.println();
    result.flush();
    result.close();
} // end of printResults()
```

8.2 ANSYS DesignModeler Automation - JavaScript

```
function planeSquareSketch(p) {
    //The base cube's square-sketching function that was derived
    //interactively:

    //Plane
    p.Plane = agb.GetActivePlane();
    p.Origin = p.Plane.GetOrigin();
    p.XAxis  = p.Plane.GetXAxis();
    p.YAxis  = p.Plane.GetYAxis();

    //Sketch
    p.Sk1 = p.Plane.NewSketch();
    p.Sk1.Name = "Sketch1";

    //Edges
    with (p.Sk1) {
        p.Ln7 = Line(0.00000000, 0.00000000, 50.00000000, 0.00000000);
        p.Ln8 = Line(50.00000000, 0.00000000, 50.00000000, 50.00000000);
        p.Ln9 = Line(50.00000000, 50.00000000, -0.00000000, 50.00000000);
        p.Ln10 = Line(-0.00000000, 50.00000000, 0.00000000, 0.00000000);
    }

    //Dimensions and/or constraints
    with (p.Plane) {
        //Dimensions
        var dim;
        dim = HorizontalDim(p.Ln10.End, 0.00000000, 0.00000000, p.Ln8.Base,
            50.00000000, 0.00000000, 24.79892761, -4.32198450);
        if(dim) dim.Name = "H1";
        dim = VerticalDim(p.Ln7.End, 50.00000000, 0.00000000, p.Ln9.Base,
            50.00000000, 50.00000000, 53.91266029, 25.42909760);
        if(dim) dim.Name = "V2";

        //Constraints
        HorizontalCon(p.Ln7);
        HorizontalCon(p.Ln9);
        VerticalCon(p.Ln8);
        VerticalCon(p.Ln10);
        CoincidentCon(p.Ln7.End, 50.00000000, 0.00000000, p.Ln8.Base,
            50.00000000, 0.00000000);
        CoincidentCon(p.Ln8.End, 50.00000000, 50.00000000, p.Ln9.Base,
            50.00000000, 50.00000000);
        CoincidentCon(p.Ln9.End, -0.00000000, 50.00000000, p.Ln10.Base,
            -0.00000000, 50.00000000);
        CoincidentCon(p.Ln10.End, 0.00000000, 0.00000000, p.Ln7.Base, 0.00000000,
            0.00000000);
        CoincidentCon(p.Ln7.Base, 0.00000000, 0.00000000, p.Origin, 0.00000000,
            0.00000000);
    }
    //Final evaluate of all dimensions and constraints in plane
    p.Plane.EvalDimCons();
    return p;
} //End Plane JScript function: planeSquareSketch
function planeCircleSketch(p, n) {
    //Plane
    p.Plane  = agb.GetActivePlane();
    p.Origin = p.Plane.GetOrigin();
    p.XAxis  = p.Plane.GetXAxis();
    p.YAxis  = p.Plane.GetYAxis();

    //Sketch
    p.Sketch2 = p.Plane.NewSketch();
    p.Sketch2.Name = "Sketch" + n + 2;
```

```
    //Edges
    with (p.Sketch2) {
        p.Cr7 = Circle(0.00000000, 0.00000000, 0.50000000);
    }

    //Dimensions and/or constraints
    with (p.Plane) {
        //Dimensions
        var dim;
        dim = RadiusDim(p.Cr7, 6.10696552, 5.78701350, 0);
        if(dim) dim.Name = "R1";

        //Constraints
        CoincidentCon(p.Cr7.Center, 0.00000000, 0.00000000, p.Origin, 0.00000000,
            0.00000000);
    }

    //Final evaluate of all dimensions and constraints in plane
    p.Plane.EvalDimCons();
    return p;
} //End Plane JScript function: planeCircleSketch
////////////// End of macro functions //////////////
//Call Plane JScript function on XY plane
var XYPlane = agb.getXYPlane();
agb.SetActivePlane(XYPlane);
var ps1 = planeSquareSketch(new Object());
agb.Regen();
//Extrude the square generated above on the XY plane
var ext1 = agb.Extrude(agc.Add, ps1.Sk1, agc.DirNormal, agc.ExtentFixed,
    50.0, agc.ExtentFixed, 0.0, agc.No, 0.0, 0.0);
ext1.Name = "ExtrudeCube"; agb.Regen();
//Import the points (2 per straight fiber) from the generated coordinates file
var fp1 = agb.FPoint(agc.FPointConstruction, agc.FPointCoordinateFile);
fp1.Name = "FiberPoints";
fp1.CoordinateFile = "C:\\Users\\Andrei\\Documents\\ANSYS_Projects\\
    StraightFibers_Cubes\\StraightFibers500nm_50umCubes_100Fibers_files\\
    FiberData_Arr2.txt";
agb.Regen();
//Generate line bodies according to the sets of points from fp1 above
LP = new Array();
var i = 1;
while (fp1.GetPoint(i, 1)) {
    LP[i] = agb.LinePt();
    LP[i].Name = "FiberLine_" + i;
    LP[i].AddSegment(fp1.GetPoint(i, 1), fp1.GetPoint(i, 2), i);
    agb.Regen();
    i++;
}
var NS_Fibers = new Array(); for (var j = 1; j < i; j++) {
    ag.b.AddSelectEdgeID(j);
    NS_Fibers[j] = ag.gui.CreateSelectionSet();
    ag.gui.ClearSelect();
}
//Generate one plane for each segment of LP1 via the "from point and normal"
//method, then draw circle at origin of plane, and then sweep circle through
//the line body.
var planeName = "FiberPlane";
for (var j = 1; j < i; j++) {
    ag.gui.ClearSelect();

    //Generate plane
    var fplane = agb.PlaneFromPointNormal(fp1.GetPoint(j, 1), LP[j].GetEdge(1));
    if (j < 10) {
    .    fplane.Name = planeName + "_0" + j;
    }
    else {
     fplane.Name = planeName + "_" + j;
    }
```

```
    agb.Regen();
    agb.SetActivePlane(fplane);

    //Generate circle sketch
    //Passes "j" to planeCircleSketch as well for numbering, otherwise Warning
    //appears when each successive plane is generated.
    var ps2 = planeCircleSketch(new Object(), j);
    agb.Regen();
    //Generate sweep
    var sweep1 = agb.Sweep(agc.Slice, ps2.Sketch2, NS_Fibers[j],
     agc.AlignTangent, 1.0, 0, agc.No, 0.0, 0.0);
    sweep1.Name = "Sweep_" + j;
    agb.Regen();
}
```

8.3 ANSYS Fluent Automation - Java and IronPython

The script below is an example of those used for automating the Fluent solver to solve multiple variations of systems. This particular script was used to automate the generation of fiberless "diffusion cubes" used in part 6.

Rather than risk making hard-to-trace syntax errors in transitioning between the various programming languages and macro APIs used by the software in ANSYS Workbench, I found the simple method of using Java to lay out a journal file with simple repetition of commands if loops were necessary (since the loops could be implemented in Java itself to save time, and simply used to copy existing commands in the macro file if multiple operations were desired).

```
/**
 * This script is used to quickly generate series of duplicates of systems in Workbench projects.
 * If the duplicated cases share geometry and will use the same general Setup scheme in
 * Fluent, it is especially helpful to define one case and then duplicate it, since the duplicated
 * copies will inherit these set parameters and they will not have to be re-assigned. When many
 * similar cases will be computed with only a few variations between them, it is much It is easier
 * to make small changes to cases already defined with some general applicable parameters than to
 * have to reselect materials, material properties, solvers, calculation activities, and so forth.
 *
 * Before running this script, ensure that your console buffer length is sufficiently long to store
 * the output of this script. After the script runs, copy the output in the console to a text file,
 * then cut the bottom portion (under the separator) and paste it in another text file. Both text
 * files should be saved with the ".wbjn" extension. The upper portion of the console output provides
 * the journal "macro" for duplicating the systems within Workbench, and the bottom half contains the
 * journal that passes commands to Fluent & automates the solution.
 *
 * Ensure that the line stating "...***Fluent Journal Below for All Systems Duplicated Above!***..."
 * is deleted (and not present in either of the .wbjn files you save!) or it could cause bugs in
 * Workbench or Fluent.
 * @author Andrei Georgescu
 **/
public class Workbench_Duplicator {
public static void main(String[] arg) {
//Set to "true" if you want to simply define the number of duplicates created,
//or false if you want to define start and end parameters for some variation
     //parameter that you are controlling. In this case, useLength was set to false
//and the duplication was controlled by values assigned to mass diffusivity
     //parameters for the species-model mixture material.
        boolean useLength = false;
//Set to "true" if Fluent journal files outlining parameters
//for duplicated cases is also desired. This will insert Recommend "true".
//Remember to copy/paste cube_macros file into each case folder!
boolean provideFluentJournal = true;
int existingID = 1; //Base system from which new ones are duplicated
int startID = 2; //Requires proper value if provideFluentJournal = true!
int length = 50; //If useLength = true, this needs to be set!
double startVal = 6;
double valIncrement = 2;
double endVal = 200;
///                                End of user input                                ///
```

```
////////////////////////////////////////////////////////////////////////////////
///                              Start of output generation                    ///
// This first portion provides a journal file to duplicate your inputs, starting from the
        // values specified earlier. Naming may be customized by changing the lines marked with the
        // comment: "//NAME_CHANGE." Both lines should be modified if the name for the duplicated
        // systems is changed.
int currentID = startID;
double currentVal = startVal;
String initial = new String("# encoding: utf-8\nSetScriptVersion(Version=\"14.0\")\n");
System.out.println(initial);
System.out.printf("system1 = GetSystem(Name=\"FFF %d\")\n", existingID);
if (useLength) {
for (int i = 1; i <= length; i++) {
System.out.println();
System.out.printf(
"system2 = system1.Duplicate(RelativeTo=system1)\n" +
"system2.DisplayText = \"0.00025_%1.1fe-11\"", currentVal);//NAME_CHANGE
System.out.println();
currentVal += valIncrement;
}//end of while loop
} else {
while (currentVal <= endVal) {
System.out.println();
System.out.printf(
"system2 = system1.Duplicate(RelativeTo=system1)\n" +
"system2.DisplayText = \"0.00025_%1.1fe-11\"", currentVal);//NAME_CHANGE
System.out.println();
currentVal += valIncrement;
} //end of while loop
} // end of if-else
currentVal = startVal; //Resets currentVal
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
// This second portion outputs the journal commands that Workbench will be passing to
        // Fluent as you run the software. If any modifications are made or more text is imported,
// it is important to remember that *all* quotation marks and all backwards slashes should
// have an extra backwards slash applied as an "escape." For example, some of the strings
// below that originally contained the two characters:   \"   already have the "\" escape
// character, and thus another two backslashes need to be added in order to prevent Java
// from removing these escape characters.
if (provideFluentJournal) {
  System.out.println();System.out.println();System.out.println();
  System.out.println(
       "##******** Fluent Journal Below for All Systems Duplicated Above! ********");
  System.out.println();System.out.println();System.out.println();
  System.out.print(initial);
  if (useLength) {
    for (int i = 1; i <= length; i++) {
System.out.println();
//Begin text for each Fluent run below
System.out.printf(
            "system1 = GetSystem(Name=\"FFF %d\")\n" +
            "component1 = system1.GetComponent(Name=\"Solution\")\n" +
            "component1.Refresh()\n" +
            "solution1 = system1.GetContainer(ComponentName=\"Solution\")\n" +
            "solution1.Edit()\n" +
            "component1.Refresh()\n" +
            "setup1 = system1.GetContainer(ComponentName=\"Setup\")\n" +
            "setup1.SendCommand(Command=\"/file/read-macros \\\"cube_macros\\\"\")\n" +
            "setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
                 "\\\"NavigationPane*Frame1*PushButton4(Materials)\\\")\")\n" +
            "setup1.SendCommand(Command=\"(cx-gui-do cx-set-list-selections" +
                  "\\\"Materials*Frame1*Table1*Frame1*List1(Materials)\\\"" +
                  " '( 1))(cx-gui-do cx-activate-item" +
                  "\\\"Materials*Frame1*Table1*Frame1*List1(Materials)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item
      "\\\"Materials*Frame1*Table1*ButtonBox2*PushButton1(Create/Edit)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-set-real-entry-list \\\"Create/Edit" +
```

```
      "Materials*Frame2(Properties)*Table2(Properties)*" +
                  "Frame9*Frame2*RealEntry3\\\" '( %1.1fe-011))\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item \\\"Create/Edit" +
      "Materials*PanelButtons*PushButton1(Change/Create)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item \\\"Create/Edit
      "Materials*PanelButtons*PushButton1(Close)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item
      "\\\"NavigationPane*Frame1*PushButton17(Run Calculation)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item \\\"Run
Calculation*Frame1*Table1*PushButton21(Calculate)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item \\\"ToolBar*General
      "Tools*write\\\")\")\n" +
"Save(Overwrite=True)\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
                  "\\\"MenuBar*PopupMenuWrite*Save Project\\\")\")\n" +
"setup1.SendCommand(Command=\"(%%cx-warning-dialog \\\"OK to close FLUENT?\\\"" +
      "#f)(cx-gui-do cx-activate-item \\\"Warning*OK\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item \\\"MenuBar*FileMenu*Close
       "FLUENT\\\")\")", currentID, currentVal);
    //
// End of text for each Fluent run
//
System.out.println();
currentVal += valIncrement;
currentID++;
}//end of while loop
} else {
while (currentVal <= endVal) {
System.out.println();
//Begin text for each Fluent run below
System.out.printf("system1 = GetSystem(Name=\"FFF %d\")\n" +
"component1 = system1.GetComponent(Name=\"Solution\")\n" +
"component1.Refresh()\n" +
"solution1 = system1.GetContainer(ComponentName=\"Solution\")\n" +
"solution1.Edit()\n" +
"component1.Refresh()\n" +
"setup1 = system1.GetContainer(ComponentName=\"Setup\")\n" +
"setup1.SendCommand(Command=\"/file/read-macros" +
   "\\\"cube_macros\\\"\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
                                    "\\\"NavigationPane*Frame1*PushButton4(Materials)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-set-list-selections
   \\\"Materials*Frame1*Table1*Frame1*List1(Materials)\\\" " +
"'( 1))(cx-gui-do cx-activate-item"
       "\\\"Materials*Frame1*Table1*Frame1*List1(Materials)\\\")\")\n" +
  "setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item"
     "\\\"Materials*Frame1*Table1*ButtonBox2*PushButton1(Create/Edit)\\\")\")\n"  +
"setup1.SendCommand(Command=\"(cx-gui-do cx-set-real-entry-list"
   "\\\"Create/Edit Materials*Frame2(Properties)*Table2(Properties)*" +
"Frame9*Frame2*RealEntry3\\\" '( %1.1fe-011))\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
          "\\\"Create/Edit Materials*PanelButtons*PushButton1(Change/Create)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
      "\\\"Create/Edit Materials*PanelButtons*PushButton1(Close)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
             "\\\"NavigationPane*Frame1*PushButton17(Run Calculation)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
 "\\\"Run Calculation*Frame1*Table1*PushButton21(Calculate)\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
 "\\\"ToolBar*General Tools*write\\\")\")\n" +
"Save(Overwrite=True)\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
  "\\\"MenuBar*PopupMenuWrite*Save Project\\\")\")\n" +
"setup1.SendCommand(Command=\"(%%cx-warning-dialog \\\"OK to close" +
"FLUENT?\\\" #f)(cx-gui-do cx-activate-item \\\"Warning*OK\\\")\")\n" +
"setup1.SendCommand(Command=\"(cx-gui-do cx-activate-item" +
                         "\\\"MenuBar*FileMenu*Close FLUENT\\\")\")", currentID, currentVal);
                  //End of text for each Fluent run
```

```
System.out.println();
currentVal += valIncrement;
currentID++;
} //end of while loop
} // end of if-else for choosing loop method
}//End if
}//end of main()
}//end of class Workbench Duplicator
```

8.4 ANSYS CFD-Post Automation - Java and IronPython

The code below was used to automate CFD-Post in order to output numerous figures and data from the various models. Note that the journal file itself is substituted by the text "`CFD-Post Macro Output Goes Here`," since the contents of the macro may be duplicated simply by interactively using the software while Workbench is recording the journal. As with the Fluent automation script, I chose to do all of the programming in Java, whereby the the scripts/automation was controlled by Java and the macro APIs were used as simple cut-and-paste commands for interfacing with the ANSYS software.

Once more, it is always essential that care be taken when assigning a macro script to a string and ensuring that escape characters are inserted where needed, because the macro functions themselves have escape characters that Java may remove upon parsing the String assignment. An easy method by which these issues may be avoided is to open the macro/journal text file inside a simple text editor (such as Notepad++) and performing a Find and Replace; first, all backslashes (escape characters) should be replaced with two backslashes: '\\', and then all quotation marks should be appended by a backslash (' \" '). This ensures that both escape characters and quotation marks will not be affected by assignment to a String in Java.

The script below saves the main macro in a file named CFD_Automator_Full.wbjn, while also splitting the journaled file into smaller files (named from CFD_Automator_Part_01.wbjn onwards), each consisting of 15 CFD-Post tasks and also containing the original header in the journal file ("`# encoding: utf-8\n" + "SetScriptVersion(Version=\"14.0\")`).

```
import java.io.FileWriter;
import java.io.BufferedWriter;
public class CFD_Automation {
public static void main(String[] args) {
String CFD_Main_String = new String("");
```

```
String FullMacro = new String("# encoding: utf-8\n" +
"SetScriptVersion(Version=\"14.0\")\n");
String filenameImg = new String("");
String filenameExp = new String("");
int initSystemID = 2;
int lastSystemID = 99;
double initSysDiffusionCoeff = 6.0;//e-011 part not included
double changeSysDiffusionCoeff = 2.0;
int part = 1;
String partStr = new String("");
String tempFileContents = new String("# encoding: utf-8\n" +
"SetScriptVersion(Version=\"14.0\")\n");
double currentSysDiffusionCoeff = initSysDiffusionCoeff;
for (int i = initSystemID; i <= lastSystemID; i++) {
filenameImg = String.format("C:/Users/Andrei/Documents/ANSYS" +
        "Projects/diffusion_cubes_files/user_files/0_00025_%1.1fe-11.png", currentSysDiffusionCoeff);
filenameExp = String.format("C:/Users/Andrei/Documents/ANSYS" +
        "Projects/diffusion_cubes_files/user_files/0_00025_%1.1fe-11.csv", currentSysDiffusionCoeff);
            CFD_Main_String =
                    "system1 = GetSystem(Name=\"FFF " + i + "\")\n" +
                    "results1 = system1.GetContainer(ComponentName=\"Results\")\n" +
                     ...
                      (CFD-Post Macro Output goes here)
                     ...
                    "Save(Overwrite=True)\n" +
                    "results1.Exit()\n\n";
                    //End of CFX_Main_String
FullMacro = FullMacro + "\n\n\n\n" + CFX_Main_String;
tempFileContents = tempFileContents + "\n\n\n\n" + CFD_Main_String;
currentSysDiffusionCoeff += changeSysDiffusionCoeff;
if ((i - initSystemID + 1) % 15 == 0) {
try {
//create the split files
if (part > 9) {
partStr = "0" + part;
}
else {
partStr = "" + part;
}
FileWriter fileW = new FileWriter("C:/Users/Andrei/Documents/ANSYS" +
                      "Projects/diffusion_cubes_files/CFD_Automator_Part_" + partStr + ".wbjn");
BufferedWriter buffW = new BufferedWriter(fileW);
buffW.write(tempFileContents);
buffW.flush();
buffW.close();
     } catch (Exception e) {
System.out.println("Couldn't write file...");
      }
part++;
tempFileContents = "";
}//End of if-else
} //End of for loop (starts above string)
try {
//create the split files
if (part > 9) {
partStr = "0" + part;
}
else {
partStr = "" + part;
}
FileWriter fileW = new FileWriter("C:/Users/Andrei/Documents/ANSYS
               "Projects/diffusion_cubes_files/CFD_Automator_Part_" + partStr + ".wbjn");
BufferedWriter buffW = new BufferedWriter(fileW);
buffW.write(tempFileContents);
buffW.flush();
buffW.close();
} catch (Exception e) {
```

```
System.out.println("Couldn't write file...");
  }
try {
//create the full file in case part-files are bugged
FileWriter fileW = new FileWriter("C:/Users/Andrei/Documents/ANSYS" +
               "Projects/diffusion_cubes_files/CFD_Automator_Full.wbjn");
BufferedWriter buffW = new BufferedWriter(fileW);
buffW.write(FullMacro);
buffW.flush();
buffW.close();
} catch (Exception e) {
System.out.println("Couldn't write file...");
  }
} //end of main()

} // end of class CFD_Automation
```

References

[1] ANSYS Inc. ANSYS Fluent Features.

[2] ANSYS Inc. ANSYS Release 14.0: FLUENT Theory Guide, 2011.

[3] ANSYS Inc. ANSYS Release 14.0: Fluent User's Guide. Technical report, Canonsburg, PA, 2011.

[4] ANSYS Inc. CFX Best Practices for Numerical Accuracy. In *ANSYS 14.0: CFX Reference Guide*. Canonsburg, PA, 2011.

[5] Jürgen Becker. Pore Scale Modelling of Porous Layers Used in Fuel Cells. Technical report, Fraunhofer Institute for Industrial Mathematics ITWM, Kaiserslautern, Germany, 2011.

[6] Siddharth Bhopte, Bahgat Sammakia, and Bruce Murray. Numerical study of a novel passive micromixer design. *2010 12th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, pages 1–10, June 2010.

[7] Lorenzo Capretto, Wei Cheng, Martyn Hill, and Xunli Zhang. Micromixing Within Microfluidic Devices. (April):27–68, 2011.

[8] Zhenhua Chai, Baochang Shi, Jianhua Lu, and Zhaoli Guo. Non-Darcy flow in disordered porous media: A lattice Boltzmann study. *Computers & Fluids*, 39(10):2069–2077, December 2010.

[9] Daehwan Cho, Lauren Matlock-Colangelo, Chunhui Xiang, Peter J. Asiello, Antje J. Baeumner, and Margaret W. Frey. Electrospun nanofibers for microfluidic analytical systems. *Polymer*, 52(15):3413–3421, July 2011.

[10] Frank a. Coutelieris and J.M.P.Q. Delgado. *Transport Processes in Porous Media*, volume 20 of *Advanced Structured Materials*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[11] Ashim Datta and Vineet Rakesh. *An Introduction to Modeling of Transport Processes*. Cambridge University Press, New York, NY, 2010.

[12] David Goldberg. What Every Computer Scientist Floating-Point Arithmetic Should Know About. *ACM Computing Surveys*, 23(1), 1991.

[13] V Hessel, S Hardt, H Lowe, and F Schonfeld. Laminar Mixing in Different Interdigital Micromixers : I . Experimental Characterization. 49(3), 2003.

[14] Volker Hessel. *Micro Process Engineering: A Comprehensive Handbook*. Weinheim: Wiley-VCH, 2009.

[15] Weizhang Huang and Robert D. Russell. *Adaptive Moving Mesh Methods*, volume 174 of *Applied Mathematical Sciences*. Springer New York, New York, NY, 2011.

[16] Wonjin Jeon and Chee Burm Shin. Design and simulation of passive mixing in microfluidic systems with geometric variations. *Chemical Engineering Journal*, 152(2-3):575–582, October 2009.

[17] K. Khoshmanesh, F. J. Tovar-Lopez, M. Nasabi, a. Z. Kouzani, S. Nahavandi, J. R. Kanwar, S. Baratchi, K. Kalantar-zadeh, and a. Mitchell. Mixing characterisation for a serpentine microchannel equipped with embedded barriers. *Proceedings of SPIE*, 7270:727005–727005–11, 2008.

[18] Lauren Matlock-Colangelo, Daehwan Cho, Christine L Pitner, Margaret W Frey, and Antje J Baeumner. Functionalized electrospun nanofibers as bioseparators in microfluidic systems. *Lab on a chip*, 12(9):1696–701, May 2012.

[19] Behnam Pourdeyhimi, Benoit Mazé, and Hooman Vahedi Tafreshi. Simulation and Analysis of Unbonded Nonwoven Fibrous Structures. 1(2):47–65.

[20] Arun Kumar Pradhan, Dipayan Das, R. Chattopadhyay, and S.N. Singh. Effect of 3D fiber orientation distribution on transverse air permeability of fibrous porous media. *Powder Technology*, 221:101–104, May 2012.

[21] Andreas Wiegmann, Jürgen Becker, Liping Cheng, Erik Glatt, and Dr. Stefan Rief. GeoDict: A software-centered approach to the simulation of thin porous media and their properties. Technical report, Fraunhofer ITWM, Kaiserslautern, Germany, 2011.